Complexity Management in Graphical Models

Doctoral Thesis

for the Degree of A
Doctor of Informatics

At the Faculty of Economics,
Business Administration and
Information Technology
of the
University of Zurich

by
Tobias Reinhard
from
Horw LU

Accepted on the recommendation of

Prof. Dr. M. Glinz
Prof. Dr. H. C. Gall

2010

The Faculty of Economics, Business Administration and Information Technology of the University of Zurich herewith permits the publication of the aforementioned dissertation without expressing any opinion on the views contained therein.

Zurich, October 27, 2010[1]

The Vice Dean of the Academic Program in Informatics: Prof. Dr. H. C. Gall

[1] Date of the graduation

Abstract

Graphical or visual representations play a central role in the software life cycle as a means to make the immaterial software more tangible and accessible. While such drawings or diagrams facilitate a "computational offloading" when reasoning about a system, the complexity of today's software systems makes them often extremely big and cluttered. One way to cope with this size and complexity is to use hierarchical and aspectual decompositions to split the models into manageable and understandable parts. Such a decomposition mechanism is the basic idea behind the ADORA approach: It uses an integrated, inherently hierarchical model together with a tool that generates abstractions in the form of diagrams of manageable complexity. The underlying complexity management mechanism combines two concepts: (i) a fisheye zoom visualization which shows local detail and its surrounding global context in one single view and (ii) a dynamic generation of different views by filtering specific model elements.

The work at hand covers the technical foundations of this complexity management mechanism. While the simplicity of the basic concept contributes largely to its appeal, the actual realization in a computer-based tool has to cope with a lot of conceptual and technical problems and trade-offs. Besides the presentation and discussion of the actual data structures and algorithms, the detailed requirements they have to fulfill are covered as well. An improved fisheye zoom algorithm that employs the concept of interval scaling and solves the problem of having a user-editable layout which is stable under multiple zoom operations builds the basis for the dynamic adaption of a diagram. This algorithm can be extended to adapt the layout if model elements are filtered to generate different views on the model. Additionally, it can be used to support the model editing by adapting the layout automatically. Since these automatic layout adjustments result in a dynamic, constantly changing diagram, the links or lines connecting model elements have to be adapted, too. As a solution to that problem, an automatic line routing algorithm that produces an aesthetically appealing layout and routes in real time has been developed. The basic data structure of this algorithm can also be used to automatically place the labels accompanying the links.

Zusammenfassung

Graphische Repräsentationen spielen im Software Lebenszyklus eine zentrale Rolle dabei, immaterielle Software fassbarer und zugänglicher zu machen. Obwohl solche Grafiken oder Diagramme das sogenannte "computational offloading" beim Verstehen eines Systems fördern, führt die Komplexität heutiger Softwaresysteme oftmals zu sehr grossen und überladenen Modellen. Eine Möglichkeit um dieser Grösse und Komplexität Herr zu werden liegt in einer hierarchischen und aspekt-basierten Dekomposition des Modells in handhabbare und verständliche Teile. Eine solche Dekomposition ist die grundlegende Idee hinter ADORA: Der Ansatz beruht auf einem integrierten, inhärent hierarchischen Modell zusammen mit einem Werkzeug, welches Abstraktionen in der Form von Diagrammen handhabbarer Grösse und Komplexität generiert. Der Mechanismus zur Handhabung der Komplexität kombiniert zwei Konzepte: a) eine sogenannte Fischaugen-Visualisierung, welche lokale Details zusammen mit dem umgebenden Kontext in einer einzigen Ansicht darstellt, und b) eine dynamische Generierung verschiedener Sichten durch das Ausblenden spezifischer Modellelemente.

Die vorliegende Arbeit beschäftigt sich mit den technischen Grundlagen dieses Ansatzes des Umgangs mit Komplexität. Obwohl die Einfachheit des Konzepts einen grossen Teil dessen Attraktivität ausmacht, tauchen bei der Realisierung in einem rechnergestützten Werkzeug eine ganze Menge technischer Probleme und Zielkonflikte auf. Neben der Präsentation und Diskussion der dafür benötigen Datenstrukturen und Algorithmen, werden auch die Anforderungen, welche diese zu erfüllen haben, behandelt. Ein verbesserter Fischaugen Zoomalgorithmus, der auf dem Prinzip der Skalierung von Intervallen beruht und sowohl die Editierbarkeit des Layouts als auch dessen Stabilität bei mehreren Zoomoperationen garantiert, bildet die Basis für die dynamische Anpassung des Layouts. Dieser Algorithmus kann, mit kleinen Anpassungen, auch dazu verwendet werden, das Layout beim Generieren verschiedener Sichten durch das Ausblenden spezifischer Modellelemente anzupassen. Zusätzlich unterstützt er den Benutzer beim Editieren des Modells durch eine automatische Anpassung. Da diese automatischen Veränderun-

gen des Layouts in dynamischen, sich ständig ändernden Diagrammen resultieren, müssen auch die Linien, welche die Modellelemente verbinden, angepasst werden. Als Lösung für dieses Problem wurde ein automatischer Linienführungsalgorithmus entwickelt, der in Echtzeit ästhetisch ansprechende Linien generiert. Die Datenstruktur dieses Algorithmus kann zusätzlich auch zur automatischen Platzierung von Linienbeschriftungen verwendet werden.

Contents

List of Figures

Part I

Introduction and Background

CHAPTER 1

Introduction

1.1 Motivation

The size and complexity of software systems have increased by several orders of magnitude since the early stages of the computing field in the 1940s. As a consequence, the management and understanding of large software systems has become one of the most fundamental challenges in the software engineering discipline. Graphical techniques such as drawings or diagrams are used as an aid to make parts of the immaterial software more tangible. However, the quantity of information is often so great that the diagrams become extremely big and cluttered. Given the size and resolution of current display screens, only parts of large models can be shown at a time. This results in problems in (i) locating a given element, (ii) interpreting an element, and (iii) relating an element to other elements, if the context of the element cannot be seen [Leung and Apperley, 1994]. The fact that a diagram does not fit on the screen complicates the task of finding specific nodes or links because the user has to scale or pan frequently to see the elements that lie outside the currently visible area [Furnas, 1986; Misue et al., 1995]. This navigation problem is aggravated by the fact that navigation in graphical models is inherently difficult: in contrast to the straight sequential reading style of text, there is no common reading style for diagrams so that the reader has to develop his own inspection strategies [Petre, 1995].

While scaling down the whole diagram makes it possible to see all model elements, it does so at the expense of losing the details: some elements, in particular labels, become unreadable. A more promising way to cope with the size and complexity is using hierarchical and/or aspectual decompositions of the model into manageable and understandable parts (in a way that follows

the basic software engineering principles of information hiding and separation of concerns). Any such decomposition technique, whether based on the hierarchy or different aspects, can fully play to its strength only if it is integrated into the visualization and navigation capabilities of a tool. However, most tools still rely on flat or practically flat models (i.e., they show all elements in one view) and support only panning and scaling for navigation. Tools that support navigation in hierarchical models visualize only one level of the hierarchy at a time (i.e., the composites and their components are shown as separate diagrams). This "explosive zooming" [Berner et al., 1998] leads to frequent "context switches" [Donoho et al., 1988] because each zoom step results in a completely new view. In contrast, what we really expect from a modeling tool is that it displays not only the views that have been drawn by the modeler, but exploits the full power of such hierarchical models by allowing arbitrarily navigation and zooming. In particular, abstraction and filtering capabilities are required. Abstractions help modelers concentrate on their focus of interest by hiding all lower levels of the hierarchy. Filtering mechanisms display only those model element the modeler is currently interested in and hide the others.

Given the poor support of current modeling tools for hierarchical and aspectual decompositions, it is no surprise that most modeling languages, including UML [OMG, 2005] as the most prominent example, rely on the principle of loosely coupled multi-diagram models as the primary means for separating concerns and decomposing large models into manageable parts. Only recently, hierarchical features have been added to UML in version 2.0. The main problem with this segregation over multiple diagrams is that it puts the intellectually demanding burden of integrating different diagrams in one coherent model entirely on the user's shoulders [Dori, 2002]. The reader of the model has to switch back and forth between different diagrams which makes the already hard navigation task even harder. Such switches occur frequently because the analysis and design process involves a constant interplay between different aspects of a system.

1.2 Goals and Contributions

The basic idea of the ADORA approach [Berner, 2002; Glinz et al., 2002; Joos, 1999; Meier, 2009; Seybold, 2006; Xia, 2005] is to solve this problem by reversing the underlying principles: it uses an integrated, inherently hierarchical model instead of a loose collection of diagrams and a tool that generates abstractions and diagrams of manageable complexity by exploiting the hierarchy and filtering model elements. This relieves the user from both tasks of manually scattering different aspects of the system over multiple diagrams and integrating them back upon inspections and manipulations. ADORA is comprised of (i) an integrated hierarchical modeling language with hierarchical decomposition and views (structure, behavior, user interaction, etc.) and (ii) a tool that allows a user to navigate through the hierarchy and show or hide model elements according to the selected view(s). This makes it possible to reduce the size and complexity of graphical models by interactively generating dynamic abstractions. This complexity management mechanism for graphical models combines two concepts: (i) a fisheye zoom visualization which shows local detail and its surrounding global context in one single view and (ii) a dynamic

generation of different views on the model by filtering specific model elements. The view generation mechanism facilitates the integration of multiple system aspects in one coherent model while keeping the size and complexity of the resulting diagram within reasonable limits.

As part of ADORA's complexity management mechanism, the aim of the work presented in this thesis is twofold. First, the development of an improved algorithm that adjusts the layout in case of zoom operations and the generation of dynamic views on a model. In contrast to existing techniques (including the one that was initially used for the ADORA tool [Berner, 2002; Berner et al., 1998]), the stability of zooming and filtering operations is one of the most important properties of our approach. Furthermore, most existing techniques do not support model editing (movement, addition and removal of model elements) very well. With our fisheye zoom technique, the user can freely navigate within the hierarchical structure of a model while the algorithm solves the problem of having a user-editable layout which is nevertheless stable under multiple zooming operations. The second aim is to relieve the user from the tedious drawing tasks that are not an integral part of modeling a system but rather supporting activities. The underlying goal is to move graphical editors which are still mostly (more or less) simple drawing tools along the path which source code editors have taken from simple text editors to integrated development environments (IDE). This comprises extended tool support during editing operations, a concept and algorithm that automatically routes the lines in a graphical model and a basic algorithm to place the labels which accompany the lines automatically. Fig. 1.1 illustrates the various problems that occur with graphical editors that provide only very basic support: nodes, links and link labels overlap and cross freely which renders the whole diagram largely unreadable. Instead of leaving the tasks of moving nodes, adjusting links and placing labels in order to achieve a readable layout to the user, the tool has to adapt the layout with a result like the one in Fig. 1.2.

In order to achieve these goals, this thesis tries to answer the following questions: What are the basic requirements for algorithms enabling a tool-supported complexity management technique? How do these algorithms look like? Which additional modeling activities have to be supported or automated by a modeling tool? What are the properties such supporting or automating techniques must have and how can the underlying algorithms look like? How does the interaction with such a modeling tool look like and are the proposed concepts really useful?

While most of the theoretical foundations of the visualization concept of ADORA have been covered in [Berner, 2002], an implementation that shows that these concepts can indeed work has not been provided. As it tries to close this gap, the presented work constitutes a *technical design* thesis. All presented concepts have been implemented as part of the ADORA tool. The implementation is used to demonstrate that the proposed techniques have the desired properties. Besides this basic validation in an implementation, we have conducted a *controlled experiment* to find out whether at all and how exactly our technique is used and how useful it is. As part of this classroom experiment with students our abstraction mechanism has also been compared to a flat model in order to find out whether there are any significant performance differences. While the problem of visualizing large structures on a small screen is widespread and the presented concepts can easily be reused in other domains, this thesis focuses on the visualization of software systems.

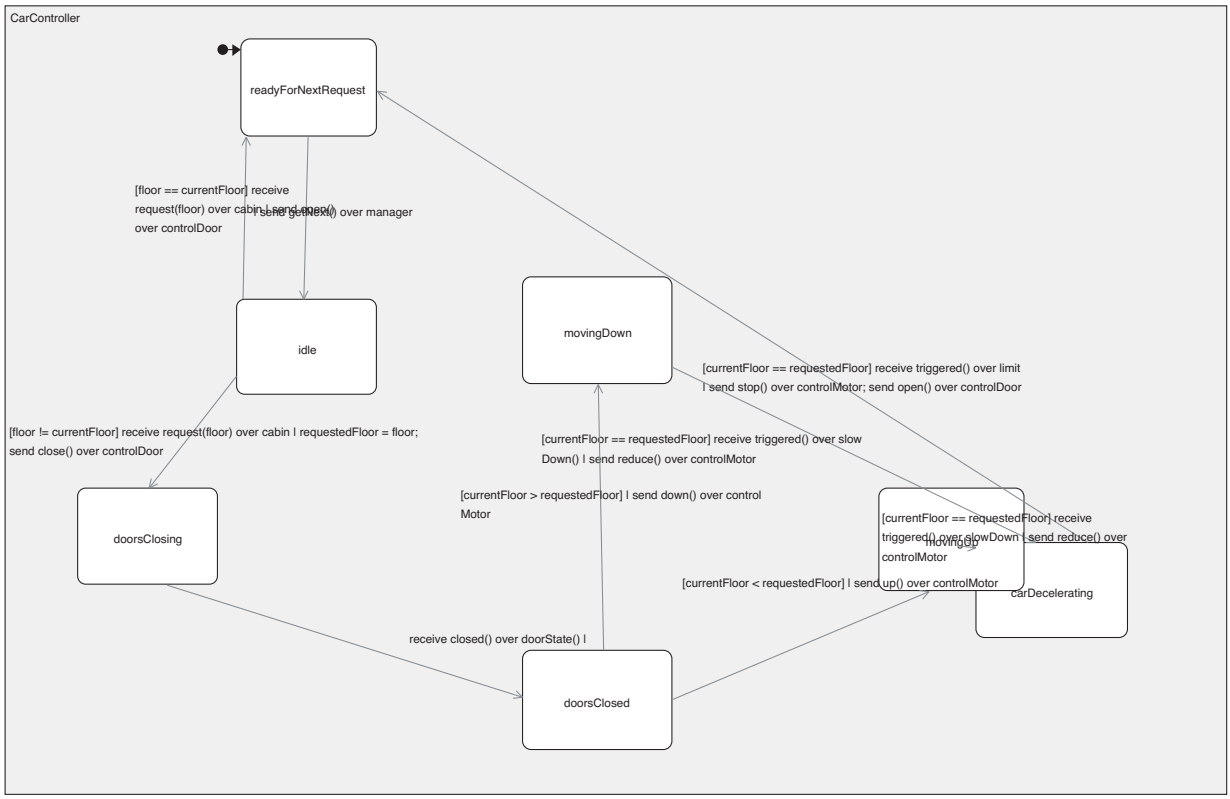


Figure 1.1: Typical layout problems resulting from only rudimentary tool support

1.3 Thesis Outline

The thesis is split into several parts. The remainder of the introductory part first sketches in Chapter 2 the modeling field in general and its role in the software engineering discipline. In Chapter 3 the arising problems with the size and complexity of models of software systems and techniques to mitigate them are discussed. Chapter 4 introduces the modeling language ADORA and shows its way to deal with the complexity and size of models. While most of the concepts of this thesis are presented independently of any specific modeling language (with the rudimentary notation of nested node-link diagrams as the only requisite), the ADORA chapter shows their use within the context of a specific language.

Our own approach to cope with the complexity and size of graphical models is the topic of the second part. The first half of it comprises fisheye zoom techniques that can be used to show both local detail and global context and different levels of detail. The properties such a fisheye zoom technique must have in order to be usable for iteratively created and maintained diagrams are introduced in Chapter 5. Existing fisheye zoom techniques are then discussed with a specific emphasis on this properties in Chapter 6. In Chapter 7 our fisheye zoom algorithm is presented in detail. Chapter 8 shows how the presented zoom algorithm can be used to dynamically generate

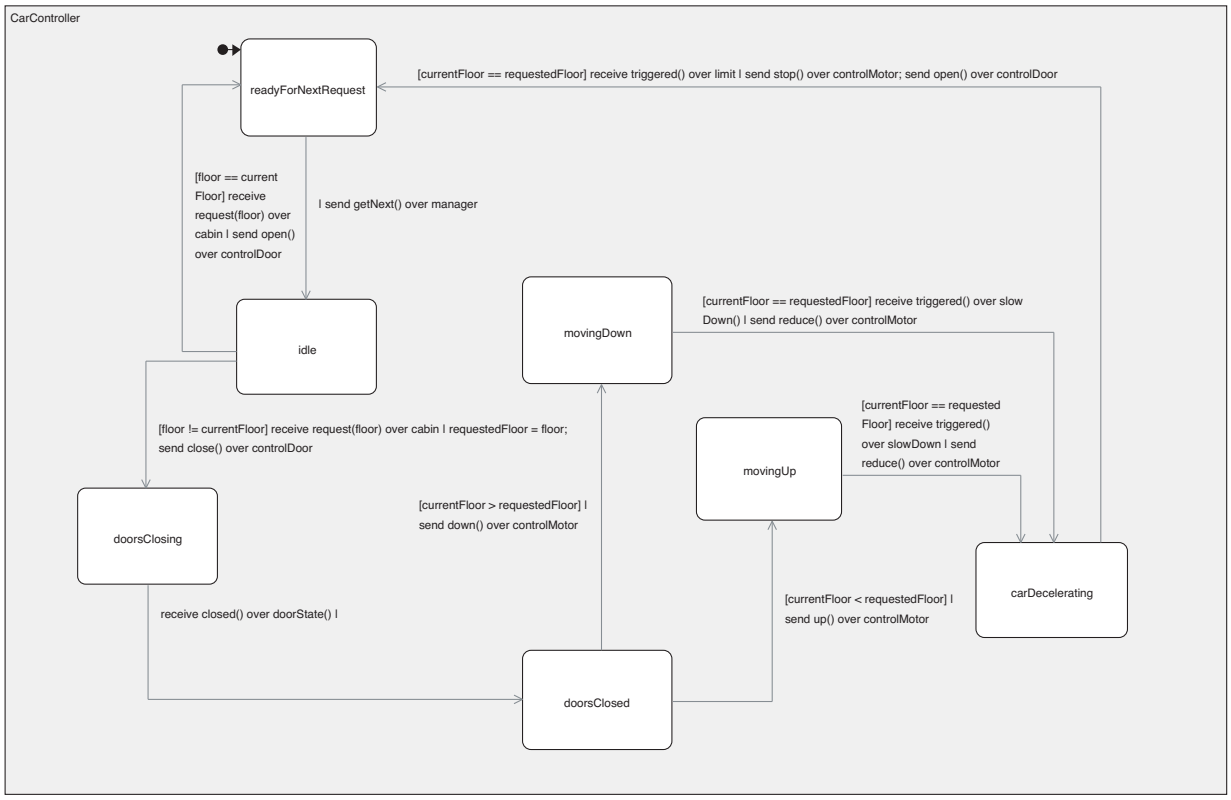


Figure 1.2: Enhanced tool support to automate layouting tasks

different views on the model by filtering specific model elements. While the previous chapter were all concerned with the complexity and size of graphical models, Chapter 9 presents an extension of our zoom algorithm to support the user during model editing. Finally, Chapter 10 discusses the presented zoom algorithm in detail. This discussion spans an evaluation against the desired properties and an assessment of the runtime and space complexity of the algorithm.

The dynamic layout that results from the application of our zoom algorithm has a big impact on the links which connect the nodes in the diagram. These links or lines are the topic of the third part. Chapter 11 sets the stage by discussing the quirks of lines in hierarchal models whose layout changes continuously. Furthermore, the requirements for an automatic line routing algorithm that works on such layouts are presented. A short overview of existing automatic line routing techniques follows in Chapter 12. Our own automatic line routing technique is introduced and discussed in Chapter 13. The second part closes with a presentation of an automatic line label placement algorithm that is built upon the line router's data structure in Chapter 14.

Part four covers the constructive and empirical validation of the presented work in Chapters 15 and 16 respectively. The conclusions and an outlook on possible future work in Chapter 17 constitute the fifth part. The last part contains the Appendix with details about the experimental validation.

CHAPTER 2

Graphical Modeling

Models and especially those that employ a graphical notation play an important role in the specification and design of software systems. This chapter gives first an overview over the use of models in general and the corresponding terminology with a special focus on graphical models. The role such models play in the software engineering field is outlined in Section 2.2. Subsequently, Section 2.3 covers the assistance computer-based tools can offer in constructing, refining and examining models. And finally, the last two sections discuss psychological aspects of graphical models, namely the role of aesthetics as well as the concepts of secondary notation and mental map.

2.1 Models and Their Representation

The ability to model allows us to interact with a complex world by reducing the vast flow of information we constantly have to cope with. Using models makes it possible to deal with some classes of problems rather than with a possibly unlimited number of individual problems. The fact that models are usually not related to an individual object or phenomena but to a class thereof (i.e., the classification abstraction) makes them an extremely powerful tool. For example, the realization that a certain class of animals rather than one single individual is dangerous has improved the humans' chance for survival significantly. Thus, models are an essential part of our everyday life. We use them - often unconsciously - to think about problems, to talk to each other, to understand phenomena and to teach. In contrast to this unconscious use of models, their construction and analysis is an explicit topic in research and engineering. Research results in

theories, which are a special kind of models, that are highly abstract and emphasize results and conclusions over obviousness. Models in engineering help in developing and understanding artifacts by providing information about the consequences of building them before they are actually made [Ludewig, 2003].

Any artifact has, according to Stachowiak [1973], to meet the following three criteria in order to qualify as a model: (i) the *mapping criterion* states that there has to be an original object (which is called the "original") or phenomenon that is mapped to the model, (ii) according to the *reduction criterion*, only some of the properties of the original are mapped to the model and (iii) the model replaces the original for some purpose to be useful which is stated by the *pragmatic criterion*. Models that are not built of concrete material are expressed in a *language*. This language consists of a set of textual and/or graphical symbols (the so-called *notation*) and the conceptual associations thereof. The rules of how to combine the symbols into valid structures are defined by the *syntax* while the *semantics* define their conceptual meaning.

2.1.1 Graphical Representations

Graphical or visual models, which employ graphical symbols as notation, occupy a central position among the different kinds of models. The saying "a picture is worth a thousand words" stands for the widely accepted argument that graphical representations are universally superior to text or any other non-graphical representation. Psychological theories explain the appeal of graphical representations with the finding that relatively large sections of static pictures or diagrams can - in contrast to the dynamic, temporally ordered nature of language - be understood in parallel. Therefore, it is possible to comprehend a complex visual structure in a fraction of a second, based on a single glance [Ware, 2004]. However, the explicit distinction between parallel and sequential representations indicates that both graphics and non-graphical representations such as text have their uses and limitations and that none is inherently superior [Petre, 1995].

Another explanation for the attractiveness of graphical representations offers the theory of external cognition [Ware, 2004] which tries to explain how resources outside the mind can be used to enhance the cognitive capabilities of the mind. Humans can interact with a rich and detailed world because the world is "its own best model". We do not need a detailed internal model of the world because whenever we want to see detail we can get it by focusing attention on some aspect or by moving our eyes to see the details in some other part of the visual field [Ware, 2004]. This principle can be exploited to solve many interesting problems that are too difficult or complex to be solved purely mentally by using cognitive tools, such as pencils and paper and increasingly computer-based tools which makes it possible to substitute seeing for reasoning. The extent of this "computational offloading" to external tools is especially large in the case of graphical representations as they enable a picture of the whole problem to be maintained simultaneously while working through the interrelated parts [Moody, 2009; Scaife and Rogers, 1996]. The emphasis on interrelated parts in this explanation indicates that graphical representations are especially powerful for relational data.

Graphical models and especially their underlying theories of cognition have gained a lot of attention during the last years due to the development of information visualization (see e.g. [Spence, 2007; Ware, 2004]) as a research field of its own. The principal task of information visualization is to facilitate the derivation of information from data by supporting the beholder in forming a mental map or image of something. Information visualization owes much of its success to the recent progress of computer graphics and the fact that visual displays provide the channel with the highest bandwidth between the computer and the human user (more information is actually acquired through vision than all the other senses combined [Ware, 2004]). While the term visualization depicts both the process of graphically representing existing data and the result thereof, we use the term graphical modeling for the process of creating a graphical representation of something. The result of this process is a graphical model, which is usually subject to change (i.e., the process is incremental).

2.1.2 Graphical Models of Abstract Phenomena

Geometric abstractions that are used to represent inherently geometric phenomena or objects in a graphical model are a very powerful tool. The floor plan of a building helps both the architect and the client to communicate and understand their intentions. Thanks to the direct geometric mapping between model and reality, contradictions and omissions become obvious [Brooks, 1987]. Scale drawings of mechanical parts and chip designs serve the same purpose. The direct relation between representation and physical reality is also the main reason why maps, which resemble miniature pictorial representation of the physical world [Tufte, 1997], are so powerful and common. But nevertheless, already the abstractions employed in maps makes it hard for many people to understand and interpret them correctly.

Many notations for graphical models of phenomena which do not have an obvious or natural graphical representation derived from their physical form[1] try to exploit the natural spatial and geographic ways of thinking to make abstract information more accessible [Perlin and Fox, 1993]. Humans intuitively tend to organize information spatially during their everyday life. For example, we are used to create stacks of similar information and place each stack at some convenient or easy remembered location on the desk [Donelson, 1978]. This spatial organization principle has also been adopted by the metaphor of the virtual desktop on the computer screen. The question is then how such graphical representations have to look like in order to tap into the human's natural spatial way of thinking.

Three-Dimensional versus Two-Dimensional Representations

The fundamental issue in any representation is how to depict a world of three or more dimensions on the "two-dimensional flatlands of paper" [Tufte, 1997]. This raises the question of whether

[1] The abstract form of the information that has to be represented is also the main point that distinguishes information visualization from other fields such as scientific visualization or geovisualization [Spence, 2007].

to use two or three dimensional representations. Obviously, a 3D representation offers one more dimension that can be used by the visualization. However, according to Gershon et al. [1998] we do not always understand when 3D is more effective. Additionally, the results of an experiment by Huotari et al. [2004] indicate that 3D is not necessarily inherently better than 2D in abstract domains and that we should therefore be cautious in making such claims. Even the increase in usable space from a 2D to a 3D display is not such a big advantage, because what is finally seen on the screen is a 2D projection of the 3D space. The fact that we can only see a 2D projection imposes a fundamental limitation that manifests itself in the fact that the usable space relates much more closely to the 2D size of n^2 than the full 3D space of n^3 since we cannot see through objects [Carpendale et al., 1997a]. Despite this, the experimental study of 3D network visualization by Ware et al. [1993] revealed that the number of errors in detecting paths through a directed graph is substantially reduced if a 3D display method is used. However, this is only achieved if the user can rotate the structure in any direction which stresses more the importance of direct interaction than that of a 3D representation itself. An additional problem in connection with the representation of software systems is that software is intangible, so that there is no obvious meaning of the third dimension (already the 2D spatial organization does not have a direct meaning).

2.1.3 Node-Link Diagrams

One of the most frequently used graphical representations is the node-link (or box and arrow) diagram. Such a diagram uses nodes to represent various kinds of entities and links to depict the relationships between these entities. The basic graphical elements of node-link diagrams are rectangles, circles or polygons that represent nodes and connecting lines depicting the links. In analogy to the phonemes which are the smallest (atomic) elements in speech recognition from which meaningful words are made, such basic graphical elements of a visual language are called graphemes [Fish and Störrle, 2007; Ware, 2004]. Graphemes are extracted by the early neural mechanisms which are in effect while looking at a graphical representation and have therefore a large influence on the perception of graphical models.

The appeal of node-link diagrams can be explained by the so-called Gestalt laws which are the result of the first serious attempt to understand the mechanisms of pattern perception undertaken in 1912 by the German psychologists Max Westheimer, Kurt Koffka and Wolfgang Kohler [Koffka, 1935]. The two most important Gestalt laws in connection with node-link diagrams are closure and connectedness [Ware, 2004]. The Gestalt law of closure states that a closed contour tends to be seen as an object and that there is a very strong perceptual tendency to divide regions of space into “inside” and “outside” the contour. This also explains why Venn-Euler diagrams are such a powerful device for displaying the relationships between sets of data. The second Gestalt law that is fundamental for node-link diagrams, connectedness, is achieved by connecting different graphical objects by lines. Connectedness is a very strong way of expressing that there exists some relationship between these objects. However, it restricts the applicability of node-link diagrams to the representation of data that is, at least partially, relational.

The use of nodes and links to depict entities and relations does only partially answer the question of how to represent abstract information graphically. The role or meaning of the (absolute or relative) location and shape of the nodes remains unclear, as there is usually no natural layout for an abstract (relational) structure [Moody, 2009]. In contrast, maps express quantities visually by location (i.e., two-dimensional addresses of latitude and longitude) and by areal extent (i.e., surface coverage) [Tufte, 1997]. Replacing the map's natural spatial scales with abstract scales of measurement that are not based on geographic analogy but still easily accessible and understandable remains a big challenge. The meaning of such geometric relationships is usually defined by the domain that is modeled.

Terminology

The theoretical background of node-link diagrams is the graph theory (see e.g. [Gross and Yellen, 1998]) which uses mathematical structures to model pairwise relations between objects. The graph theory has a widely accepted terminology: A *graph* G in this context is an ordered pair containing a set of *vertices* V and a multiset of *edges* E:

$$G = \langle V, E \rangle \tag{2.1}$$

where E contains unordered pairs of vertices which are not necessarily distinct, so that:

$$E \subseteq V \times V \tag{2.2}$$

Certain restrictions on the relations that are represented by the edges yield special classes of graphs that are of particular interest for certain applications. One the most frequently used type of graphs in computer science is the tree [Knuth, 1997]. A *tree* is a graph in which any two vertices are connected by exactly one path. A *path* in a graph is a sequence of vertices such that from each of its vertices there is an edge to the next vertex in the sequence. Alternatively, any connected graph with no cycles is a tree, where a *cycle* is a path such that the start and end vertex are the same. Two vertices U and V are *connected* if the graph contains a path from U to V. Consequently, a graph is connected if each pair of vertices in it is joined by a path. The specific topological restrictions and the broad use of trees has produced a whole terminology specifically for trees. Each vertex in a tree has an edge to exactly one *parent*, except for the *root*, which has no parent. Conversely, each vertex can have (i.e., be connected to) any number of *children*. A generalization of the parent-child relationship is the *ancestor-descendant* relationship. Vertices that have the same parent are *siblings* and a vertex that has no children is called a *leaf*. The *level* of a vertex in a tree is a measure of its distance from the root and is defined as follows: (i) if vertex X is the root of the tree, its level is 1 and (ii) if vertex X is not the root, its level is 1 + its parent's level. The *height* of a tree is the maximum level of any vertex in the tree. If a graph that has no cycles is not connected (i.e., a disjoint union of trees), it is called a *forest*.

Basic graphs can only represent binary relations on a set of elements. A *hypergraph* which is an extension of a basic graph can additionally express one-to-many relationships. An edge no longer connects a pair of nodes, but rather a subset thereof. A hypergraph H is still an ordered pair $H = \langle V, E \rangle$ where V is a set of vertices but E is now a set of non-empty subsets of V called *hyperedges*:

$$E = \mathcal{P}(V) \setminus \{\emptyset\} \tag{2.3}$$

where $\mathcal{P}(V)$ is the power set of V. Hyperedges are arbitrary sets of nodes, and can therefore contain an arbitrary number of nodes. Another extension to basic graphs that lies somewhere in the middle between simple graphs and hypergraphs are *nested graphs*. The vertices of such a nested (or hierarchically clustered) graph [Noik, 1994] can contain other vertices. A nested graph N is defined by

$$N = \langle V, E, C \rangle \tag{2.4}$$

where C is a set of containment relations which are restricted to be hierarchical (i.e., C is a tree or a forest of trees):

$$C \subseteq V \times 2^V \tag{2.5}$$

The graphs as discussed so far are abstract, non-graphical structures. A *diagram*[2] or *layout* is a graphical representation of such a graph. Vertices are represented by *nodes* and edges are depicted by *links*, which results in node-link diagrams. We use the terms node and link to distinguish them from the abstract graph theoretical concepts of vertex and edge. The one-to-many relations in hypergraphs are more difficult to draw and therefore tend to be studied using the terminology of set theory rather than the more pictorial descriptions of graph theory [Harel, 1988]. Thus, graphical representations of hypergraphs typically draw from the Venn-Euler diagrams that are used to represent set relationships. Similarly, the containment edges of nested graphs are often displayed as boxes that fully enclose the nodes associated with the nested vertices.

2.2 Graphical Models in Software Engineering

Two of the main reasons why developing software is difficult are the complexity of the problem at hand and the intangible (or invisible) nature of the software itself [Brooks, 1987]. Models play an important role in coping with these two problems and are therefore found in almost all areas and applications of software engineering[3]. We limit ourselves here to prescriptive system

[2] The terms diagram and graphical model are used interchangeably in this thesis.

[3] We follow here the definition of the IEEE [1990] which defines software engineering as "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software".

models (i.e., formal or semi-formal models of computer-based systems) and do not consider process models or process maturity models.

Models are a means of communication between customers, developers and others involved in the software engineering process. Graphical models or diagrams are often seen as especially facilitating the communication about software because they represent some aspects of the intangible software in a graphical and therefore more tangible way [Moody, 2006; Purchase et al., 2002] (even though some authors, such as Brooks [1987], claim that software is essentially unvisualizable). A frequent explanation why it is much easier to work with externally represented objects is that the capacity of the working memory is limited to only three or four objects [Ware, 2004]. However, the communication effectiveness of graphical models is just an assumption, even though a very widely accepted one [Petre, 1995; Scaife and Rogers, 1996]. In fact, a large number of diagrams do not communicate effectively and often act as a barrier rather than an aid to the user-developer communication. One of the reasons is that most graphical modeling languages are built on arbitrary conventional representation [Ware, 2004] and not on sensory symbols. Symbols of an arbitrary conventional representation can be swapped or exchanged without making any difference as long as they are used consistently. Such representations, like for example the words of any spoken or written language, are by definition socially constructed and have no natural or obvious meaning. They have the following characteristics: They are hard to learn (because the graphical codes of the alphabet and the rules of combination must be laboriously learned), easy to forget, embedded in the culture and domain, formally powerful and capable of rapid change. In contrast, sensory symbols such as a stick figure to represent a person or connecting lines to depict a relation do not require special education or training and are intuitively clear to most people. Moody [2009] proposes the "semantic transparency" to assess the extent to which the meaning of a symbol can be inferred from its appearance.

A second reason for the communication problem of diagrams is their complexity (i.e., the number of diagram elements and their interrelationships) and the lack of explicit complexity management mechanisms [Moody, 2006]. Models are a powerful tool to reduce the complexity of the represented phenomena or system because they only show the currently relevant information and abstract from the rest (cf. the reduction criterion in Section 2.1). Graphical models offer an additional possibility to reduce the number of symbols by representing some informations implicitly (by for example hierarchically nesting symbols) rather than explicitly. However, the complexity remains a problem if hundreds of symbols are shown at once. This complexity of graphical models is discussed in detail in Chapter 3. But despite these problems, graphical models have a long history in the development of software and are an integral part of todays software engineering.

Despite the central role of graphical models within the software engineering discipline, the role of their visual representations have been largely ignored or undervalued both in research and language design [Moody, 2009]. Notations are developed and evaluated exclusively at the level of their semantics with their visual (or concrete) syntax being completely ignored. However, research in diagrammatic reasoning has shown that the form of a representation has an equal, if not greater, influence on how it facilitates human communication and problem solving as its content [Moody, 2009].

2.2.1 A Short History of Graphical Modeling Languages

The popularity and importance of graphical models and especially node-link diagrams in software engineering is illustrated in the following by a short history of graphical modeling languages (without any claim for completeness). The history of graphical modeling languages dates back to the beginning of computer sciences. Probably the first widely known graphical notation is the *Flowchart* which was developed by Herman Goldstine and John von Neumann at Princeton University in late 1946 and early 1947 [Goldstine and von Neumann, 1947]. This basic idea of documenting process flow goes back to previous work in other fields such as industrial engineering. A flowchart is a node-link diagram representing an algorithm or a process by showing the steps as boxes of various kinds and their order of execution by directed links.

The focus of the *Petri net* notation which was published by Carl Adam Petri in his doctoral thesis [Petri, 1962] lies on modeling the concurrent behavior of distributed systems (Petri invented the notation years before its publication to describe chemical processes). Petri nets have two kinds of alternating nodes, places and transitions, which are connected by directed links. Places may contain any number of tokens and a transition may fire whenever there is a token at all places that have a directed link to it; when it fires, it consumes these tokens, and places them at the places it has a directed link to. In 1967, Taylor Booth [Booth, 1967] introduced the *state diagram* to describe the behavior of a system. A state diagram describes the possible states of a system or object as events occur. States are represented by nodes and state transitions by the links between these nodes.

The *Nassi-Shneiderman diagram* or Structogram was developed in 1972 by Isaac Nassi and Ben Shneiderman [Nassi and Shneiderman, 1973] as a graphical notation for the structured programming paradigm. Hence, it can be seen as the graphical answer to Edsger Dijkstra's call for the abolishment of the GOTO statement from high-level languages in the interest of improving the quality of the code [Dijkstra, 1968]. Nassi-Shneiderman diagrams use nested boxes to represent subproblems (i.e., the structure is represented by the nesting and position of the boxes without the need of any links). In contrast to Flowcharts which gained much of their popularity from the possibility to easily express GOTOs or jumps, such constructs can not be represented in Nassi-Shneiderman diagrams. The notation was popular during the 80's but is nowadays only rarely used because its abstraction level is too strongly tied to structured program code and modifications usually require the whole diagram to be redrawn. The *HIPO* (Hierarchy plus Input-Process-Output) technique, developed by IBM in 1974 for planning and documenting computer programs, consists of a hierarchy chart that graphically represents the program's (hierarchical) control structure in a tree and a set of Input-Process-Output charts that describe the functions performed by each module on the hierarchy chart [IBM, 1974]. In 1975, Michael A. Jackson published the *Jackson Structured Programming* (JSP) method for structured programming [Jackson, 1975] that is based on the correspondences between data stream and program structure. Jackson's method structures programs and data in terms of sequences, iterations and selections and represents the result graphically in a tree.

Peter Chen's *Entity-Relationship Model* (ERM), published in 1976, directed the focus of sys-

tem modeling from processes and system behavior towards data [Chen, 1976]. Its conceptual data model is represented in entity-relationship diagrams which depict the entities, relations and attributes as different kinds of nodes and connects them by links. In contrast to the entity relationship models that target mainly data-driven information systems, the *Specification and Description Language* (SDL), first published in 1976 by the International Telecommunications Union and continuously extended and adjusted over the time [ITU-T, 2000], is mainly used to describe the behavior and structure of distributed, event-driven, real-time systems. The graphical representation of SDL (SDL/GR) consists of a hierarchy of different diagrams that follow the hierarchical decomposition of a system. The top level describes the system with a set of blocks (nodes) that communicate over channels. Each of these blocks is specified by a set of processes (nodes) and connecting signal routes (links) in a diagram of its own. On the next lower level, a process is described by an extended finite state machine. The procedures of this state machine can be further divided into a set of procedure diagrams. SDL originally focused on telecommunication systems, but is currently used in additional areas including process control and real-time applications in general. The language has formal semantics, so that it can be used for code generation and simulation.

The term "structured" was the buzzword of the second half of the seventies. In 1975, Edward Yourdon and Larry L. Constantine published their *Structured Design* (SD) method [Yourdon and Constantine, 1975] that uses so-called structure charts as graphical notation to describe the relations (links) between modules (nodes). Roughly at the same time, Douglas T. Ross developed in 1976/77 the *Structured Analysis and Design Technique* (SADT) [Ross and Schoman, 1977] which uses two types of diagrams, activity models and data models. SADT offers building blocks (nodes) to represent entities and activities, and a variety of arrows to relate these boxes by links. Blocks can be decomposed in a top-down approach so that each block is described in a diagram of its own. The different kinds of relations (links) are distinguished by the side they enter or leave a block. Finally, Tom DeMarco's *Structured Analysis* (SA) [DeMarco, 1979] uses dataflow diagrams (DFD) to describe a system. A dataflow diagram consists of the nodes activity (or process), store and terminal that are connected by links, which represent the flow of data. Each activity can be described in a dataflow diagram of its own which results in a DFD hierarchy.

Two of the biggest problems of state diagrams, the combinatorial explosion of the number of states in models with a lot of parallelism and the lack of the possibility to hierarchically decompose large models, have been addressed in 1987 by the *Statechart* notation of David Harel [Harel, 1987]. In addition to state diagrams, the statechart notation allows the modeling of superstates, concurrent states, and activities as part of a state. A state in a Statechart can either be elementary, hierarchically decomposed into another Statechart or into multiple parallel statecharts. While the original finite state machines are disjunctive (i.e., the machine can only be in one of all the possible states at once), a Statechart can be in two or more states concurrently.

The abundance of methods and notations that came along with the next paradigm shift to the object-oriented software development at the beginning of the 1990s resulted in the call for a shared graphical language of description and specification. Such a communication medium for both humans and software tools has been a central part of other, admittedly more mature, engi-

neering disciplines such as civil or mechanical engineering for decades. The answer of three of the more prominent methodologists of these years was the *Unified Modeling Language* which was published as first draft in 1996.

2.2.2 The Unified Modeling Language

The Unified Modeling Language (UML) was the byproduct of the failed attempt by the "Three Amigos" James Rumbaugh, Ivar Jacobson and Grady Booch to agree upon a uniform method for object-oriented software development. UML 1.1 has been accepted as the standard modeling language by the industry consortium Object Management Group (OMG) in 1997. Thereafter, the OMG has supervised its development and has been astonishing successful in establishing UML as the de-facto industry standard for the object-oriented modeling of software systems. After several minor revisions (UML 1.2, 1.3, 1.4 and 1.5) to fix shortcomings and bugs, a major revision resulted in version 2.0 [OMG, 2005] on which the following discussion is based. UML provides a collection of several more or less loosely coupled graphical notations that facilitate the construction of requirements and design models. The language employs thirteen (node-link) diagrams to represent eight different views on the system. As shown in Fig. 2.1 the diagrams are divided into three categories (indicated by the names in italics in Fig. 2.1).

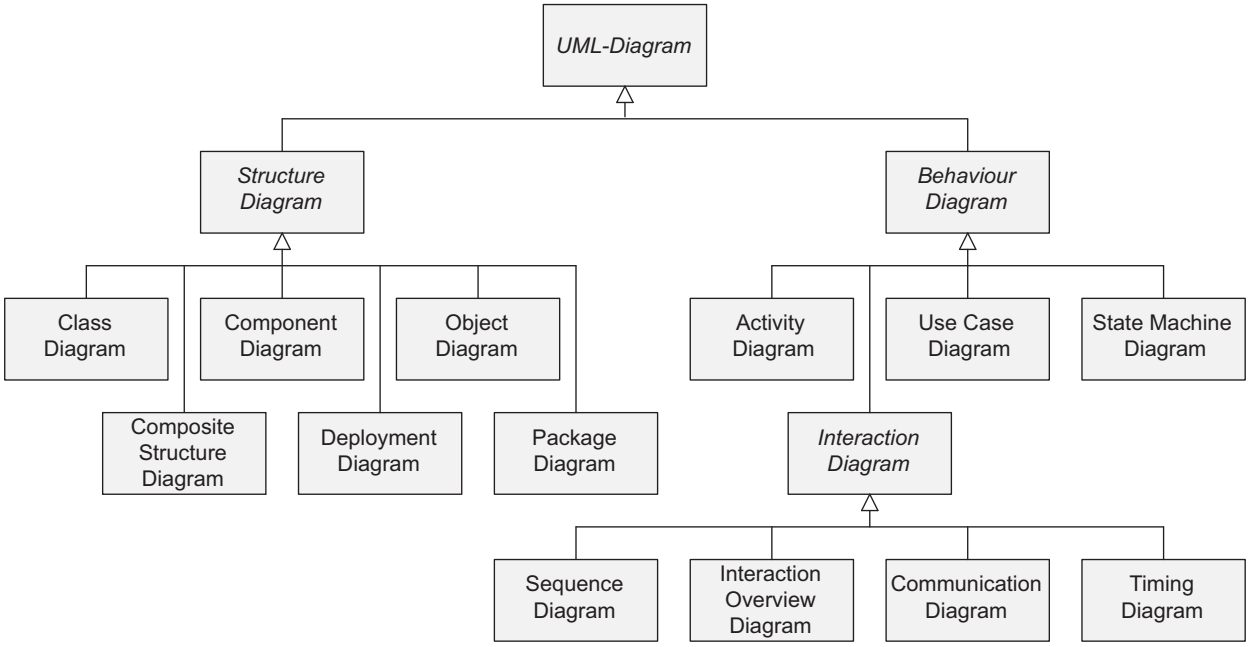


Figure 2.1: Hierarchy of UML 2.0 diagrams, represented as UML class diagram [OMG, 2005]

Six diagram types represent the static structure of the application, three general types of behavior, and the last four show different aspects of interaction. At the heart of the UML lies the *class*

diagram that specifies the static structure of the system by showing the system's classes, their attributes and methods together with the relationships between classes and among their instances. The remaining structure diagrams are the following: A *component diagram* shows how a system is split up into components and depicts the dependencies between these components. An *object diagram* shows a complete or partial view of the structure of the system at a specific moment in time. This snapshot focuses on some particular set of object instances and attributes, and the links between these instances. The *composite structure diagram* shows the internal structure of a class and the collaborations which can take place in this structure. A *deployment diagram* can be used to model how the system's components are deployed on the hardware, and the associations between the components. Finally, a *package diagram* depicts how a system is split up into logical groupings by showing the dependencies among these groupings. This very brief overview over the structure diagrams of UML already indicates that the different diagrams and the underlying concepts overlap. Therefore, it is often difficult to clearly separate UML's concepts from each other.

The behavior diagrams emphasize the dynamic behavior of the system being modeled. An *activity diagram* shows the overall flow of control similar to flowcharts but with the additional possibility to model parallelism. The UML *state machine diagram* is essentially a Harel statechart with standardized notation. *Use case diagrams* present a graphical overview of the functionality of a system in terms of actors, their goals (represented as use cases), and some rudimentary dependencies between these use cases and actors. As a subset of the behavior diagrams UML contains the following interaction diagrams that emphasize the flow of control and data between different entities of the system: A *sequence diagram* shows how a set of messages between entities are arranged in time sequence while a *timing diagram* is a a special form of such a sequence diagram with the focus on timing constraints. *Interaction overview diagrams* are a kind of activity diagram in which the activities are fragments of sequence diagrams. And finally, a *communication diagram* shows, similarly to a sequence diagram, the interactions between objects or parts. Unlike a sequence diagram, a communication diagram explicitly shows the relationships between elements but does not represent the time as a separate dimension.

UML has met with a lot of criticism over the years of its existence. The language is frequently criticized for the complexity and size of its notation (see e.g. [Dori, 2002]). The UML 2.0 superstructure standard and its complementing infrastructure standard span together more than 900 pages. The language contains, as already visible in the above description, many diagrams and constructs that are often redundant. Additionally, a lot of the constructs are rarely used in practice. The size and complexity of UML impedes its understandability as empirical evidence [Nordbotten and Crosby, 1999] indicates that notations with a simpler graphic syntax are easier to interpret. Because of its large number of symbols and keywords, UML often looks as complicated as a full programing language even though it claims to be accessible to a broader audience than any programing language. Another problem concerning the graphical notation of UML is that a lot of its symbols are overloaded [Fish and Störrle, 2007]: The same grapheme (cf. Section 2.1.2) is often used for two or more different concepts (e.g., the white or empty diamond in UML represents a decision/merge in activity diagrams, but also an aggregation in class diagrams). This

overloading is used to reduce the number of graphemes in the language but it hinders the understandability and learnability as the context (e.g., the current diagram) has additionally to be taken into account to resolve the arising ambiguity.

UML has also often been criticized for the lack of possibilities to hierarchically decompose models so that it can also be used to describe large and complex systems (see e.g. [Glinz et al., 2002; Kobryn, 2004]). It owes this deficiency to the entity-relationship notation [Chen, 1976] from which UML (like most of the object-oriented modeling notations) has evolved and which does not know a hierarchical decomposition either. The situation has improved with the release of UML 2.0 which allows a hierarchical decomposition of some structural and behavioral constructs. Such hierarchical decompositions of graphical models to manage their increasing complexity is discussed in detail in Section 3.3. Another area of concern is the often unclear or missing semantic definition of language constructs in the standard [Glinz, 2000]. In contrast to any formally defined programming language, UML is more an informally defined set of conventions that are used in different ways appropriate for the current problem at hand.

2.2.3 Graphical Programming and Model-Driven Engineering

The idea of applying graphical models not complementary to textual programming languages but using them exclusively to program graphically has risen up for the first time in the 1980s as part of the effort summarized by the then buzzword of "computer-aided software engineering" (CASE). The focus of CASE was on developing methods and tools that enabled software developers to express their designs in terms of general-purpose graphical programing representations such as state machines or flowcharts. But apart from a few specific domains such as the Matlab, Simulink, Stateflow stack in the automotive and avionics field, graphical programming had relatively little impact on the commercial software development. The aim of graphical programing was, like of any new software development paradigm, to take the developer further away from the machine details by raising the level of abstraction. A raise in the level of abstraction is the usual answer to a "software crisis" which regularly emerges as soon as the current paradigm of programming is no longer sufficient to handle the complexity of the problems developers are asked to solve [Riel, 1996]. However, most graphical programing languages do not increase the level of abstraction but employ just a different representation of computer programs than textual programing languages.

In fact, graphical representations of computer programs are often harder to understand and maintain than the equivalent representation in a high-level textual programming language. This is due to the fact that many of the cognitive operations that are required for current (sequential) programming have more in common with processing natural language than with visual processing. The source code of computer programs is actually a linear (one-dimensional) data type [Shneiderman, 1996]. Ware [2004, p. 302] attributes the failure of flowcharts to their lack of commonality with natural language (even though the lack of structure may have been as important [Brooks, 1995]). The parallel nature of our visual system explains also why graphical representations are

more frequently used to depict structural informations rather than dynamic (sequential) information such as processes or flows. As a consequence, textual and graphical representations have both their advantages and disadvantages [Petre, 1995]. Thus, hybrid approaches that take the best of both worlds by presenting and manipulating structure graphically (e.g., by representing modules as nodes connected by links) and the detailed procedures or methods using text are often more effective.

The idea of replacing textual programming languages with mostly graphical models has again gained a lot of popularity recently due to the advent of the so-called "model-driven engineering" (MDE) (see e.g. [Schmidt, 2006]). MDE spans a set of different software development approaches that use models as the primary form of expression. The focus on models is meant to avoid the problem of the current use of models in which diagrams are rarely in sync with the implementation during later stages of a project. Contrary to previous attempts, MDE emphasizes the use of domain specific modeling languages (DSML) instead of general-purpose notations. The focus on domain specific languages stems from the observation that especially visual languages are inherently domain specific and that an attempt to move beyond the domain level abstraction results in large and complex visual structures that are impossible to comprehend (as shown by the history of flowcharts or UML). The MDE proponents claim that the use of domain specific languages together with (the other buzzwords of) reusable class libraries, application frameworks and middleware platforms raises the level of abstraction available to software developers. As with any new approach that claims to reduce the complexity by an order-of-magnitude, the question remains whether the approach does not abstract away the essence of the software while trying to reduce its complexity (which is often achieved by simply decreasing its expressiveness) since the complexity is an inherent property of software [Brooks, 1987].

The currently best known MDE initiative is the Model-Driven Architecture (MDA) of the Object Management Group [OMG, 2003]. The MDA approach separates the specification of a system's functionality from the specification of the implementation of that functionality on a specific technology platform. The result of this separation are platform independent models (PIM) and platform specific models (PSM). Transformations between these models should, at least partially, be automated. The PIMs and PSMs are both intended to be expressed in OMG's Unified Modeling Language (cf. Section 2.2.2), using its profile and stereotype mechanisms to specialize and extend the language for different contexts.

2.3 Tool Support for the Modeling Process

The process of creating a graphical model of a software system is highly iterative as the model is build incrementally step by step in collaboration with different stakeholders. Computer-based tools are especially well suited to deal with changes and refinements that occur frequently in iterative processes. They make it easy to store, change and share electronic models. Most of the past and current modeling tools provide exactly these basic functionalities. These tools are essentially drawing programs that employ a palette consisting of the specific symbols that constitute

the notation of the language and enforce some of the basic constraints of its syntax. However, computer-based tools can do a lot more than just replace the paper with a screen and the pen with a mouse, especially if they exploit the inherent structure and semantics of graphical models. The possibility to directly interact with the presented information is the principal feature that separates modern computer-based tools from paper based solutions [Spence, 2007]. Objects on the screen should be active and not just a blob of color, so that they are capable of displaying more information as needed, disappearing when not needed, and accepting user commands to help with the thinking process [Ware, 2004]. Unfortunately, the support to interact with a model by exploiting its structure and semantics is rather limited in todays graphical modeling tools. The best one can hope for are the very basic semantics of node-link diagrams which are supported in so far that links move with the nodes they connect.

Computer-based tools can ease the work with graphical models but they also employ specific problems that do not occur if models are drawn on paper. Computer screens are getting bigger but are still quite small and have a poor resolution compared to the one that is possible on paper. Additionally, a computer screen is often filled with scroll bars, tool bars and other computer administration stuff that is not needed on paper. According to Hornbæk and Frøkjær [2003], the following problems occur while reading electronic (text) documents: a cumbersome navigation, lack of overview of the document, lower tangibility of electronic documents compared to paper, unclear awareness of the length of the documents, lower reading speed caused by the poor resolution of most screens and fatigue if reading for extended periods of time. Thus, paper and computer-based tools are complementary: computers are good at selecting, organizing and customizing information while paper makes the high-resolution information visible in portable, tangible and permanent form [Tufte, 1997].

2.3.1 Automatic Graph Drawing Algorithms

While computer-based modeling tools offer only moderate assistance for the creation and modification of graphical models, automation plays a much bigger role in the automatic drawing of abstract graph structures as diagrams. Automatic graph drawing algorithms take a relational graph structure as input and produce a diagram (i.e., they define the layout of the diagram) to visualize the information embodied in the graph. A multitude of such algorithms have been developed (see [Battista et al., 1994] for an overview). The metrics to measure the quality of these algorithms are the degree to which they optimize certain aesthetics (e.g., minimal number of link crossings, maximal symmetry or minimal number of bends in links) which should help the human reader to understand the information embodied in the graph and the algorithm's computational efficiency. In the software engineering field, automatic graph drawing algorithms are widely used by reengineering tools to visualize the static structure of existing software systems.

Applying automatic graph drawing algorithms in a specific domain, such as the visualization of the structure of a software system, entails two main problems. First, algorithms that are designed for abstract graph structures do not necessarily produce useful visualizations of semantic infor-

mation because they cannot take the semantics of the domain the graph is used in into account. But, exactly this meaning of the nodes and links is the most important criterion for the layout. A good layout is to a large extent based on high level semantics which cannot be formalized at present so that they cannot be employed in a graph drawing algorithm [Ware et al., 1993]. Those conventions and common sense that can be formalized have to be coded explicitly into the algorithm. For example, reengineering tools often produce UML class diagrams of parts of a software system to show the system's structure. Class diagrams are usually drawn with the convention that a superclass is drawn above its subclasses. This simple convention is already quite hard to achieve with an automatic graph drawing algorithm. The second big problem is that the user has usually no or only very little influence on the layout that is produced by an automatic graph drawing algorithm. The layout is completely defined by the algorithm and it is impossible for the user to adapt the layout to his personal preferences.

2.3.2 Layout Creation and Layout Adjustment

An important distinction between domains that employ automatic graph drawing algorithms and the manual incremental creation of graphical models is the process that yields the diagram. Automatic layout algorithms take an abstract structure as input and produce, in one step (i.e., in batch mode), a layout for the entire graph. Small changes in the abstract structure (e.g., the addition or removal of just one link) can have a tremendous influence on the resulting layout because the layout is calculated from scratch after every change. Hence, automatic graph drawing algorithms cannot be used for incremental layouts because the diagram is totally rearranged each time the user changes something. In contrast to the graph drawing domain, the graphical modeling domain is not dealing with the creation of a complete layout from an abstract representation, but with the adaption of an existing layout to relatively minor changes. Thus, the goal is not a globally optimal layout starting from a randomly chosen initial setup, but instead to find the nearest local optimum starting from a relatively good initial position [Misue et al., 1995].

Misue et al. [1995] distinguish between "layout creation" and "layout adjustment" for the two different modes to produce a diagram. Layout creation is the calculation of a layout for an abstract structure in one step. In the layout adjustment mode, changes are constantly made. A layout that has been spoiled in a local manner by such a change has to be adjusted accordingly without totally rearranging the overall layout. Freire and Rodríguez [2006] use the terms "fixed" and "incremental" layout, while Miriyala et al. [1993] distinguish between a "static" and "dynamic" mode of automatic graph drawing. Bridgeman and Tamassia [2000] go one step further by distinguishing between "incremental graph drawing" where nodes are added one at a time and the more general case of "interactive graph drawing", in which any combination of node/link insertion and removal is allowed at each step.

2.4 Aesthetics in Diagrams

Even though the automatic graph drawing algorithms can usually not be used directly by a modeling tool, they raise with their attempt to optimize certain aesthetic criteria the question about the role of aesthetics in graphical models. These aesthetics play also an important role for other tasks beside the automatic layout. For example, the best way to route the links in a node-link diagram is largely defined by the underlying aesthetic criteria. The graph drawing literature lists a large number of aesthetics which an algorithm should optimize to help the human reader in understanding the information embodied in the underlying graph structure (see e.g. [Tamassia et al., 1988] for general graphs or [Sindre et al., 1993] for nested graphs). However, there has been little attempt to use empirical methods to determine whether graphs that are laid out according to specific aesthetic principles are in fact easier to understand.

Two exceptions are the work by Ware et al. [2002], which provides empirical support for the claim that the number of bends in a link and the number of link crossings degrade performance on the task of finding the shortest path between two nodes, and the experiment by Purchase et al. [2002] to investigate the merit of automatic graph layout algorithms with respect to human use. As part of their experiment, Helen Purchase and her colleagues measured the effect of individual aesthetics on human performance (i.e., response time for each question and the correctness of the answer) with respect to the ability of subjects to answer questions about the structure of an abstract graph. The questions were chosen so that they measure only the understandability of the semantics of a node-link diagrams (i.e., the concept of nodes that are connected by links) independently of a specific domain. Example questions were the length of the shortest path between two given nodes or the number of nodes that have to be removed from a graph in order to disconnect two given nodes completely. The five investigated aesthetics were: (i) minimizing the total *number of bends* in the links, (ii) minimize the *number of link crossings* in the diagram, (iii) maximize the *angle between links* extending from a node, (iv) maximize the *orthogonality* by fixing the nodes and links on an orthogonal grid and (v) displaying a *symmetrical view* of the graph if possible. The result of the experiment with 55 individuals indicated support for reducing the number of links crossings and bends in the links, and increasing the display of symmetry. No support was found for maximizing the minimum angle or increasing orthogonality.

A second set of experiments aimed to investigate how the semantic domain of the graph drawings affects which aesthetic criteria need to be emphasized. UML class and collaboration diagrams where used as the semantic domain. The users' preference was used as the method of usability measurement in these experiments. The experiments resulted in the following priority order of aesthetics for both UML class and collaboration diagrams: (1) minimize the number of edge crossings, (2) maximize orthogonality, (3) explicitly indicate the direction of information flow on the links, (4) minimize the number of bends in the links, (5) use only horizontally aligned labels, (6) minimize the physical width of the layout and (7) employ only one font type for labels rather than using different fonts for different types of labels. This priority order of aesthetics suggests that the semantic domain of the graph drawings affects the aesthetic criteria that should be emphasized. For example, the orthogonality of the drawing was not a significant aesthetic criteria

for abstract graphs but seems to be important for UML class and collaboration diagrams. This is a clear sign that algorithms that are designed for abstract graph structures will not necessarily produce useful visualizations of semantic information. Additionally, the experiments revealed the difficulties to exclude confounding factors from the study. The aesthetic features that were investigated have been selected from a large set (see e.g., [Coleman and Parker, 1996]). Out of these, the direction of information flow has not been targeted as an aesthetic feature in the beginning but had to be added later in order to explain some unexpected results.

2.5 Secondary Notation and Mental Map

The difficulties in drawing diagrams automatically do not only stem from the neglect of the semantics of the modeling language but also from layout cues that are often used in diagrams to exhibit structures and relationships. These cues are typically not formally part of the language definition, but belong to a so-called "secondary notation" [Petre, 1995]. This secondary notation uses layout and perceptual cues (e.g., clustering, white space, relative or absolute position and/or size) to clarify information or give hints to the reader. Fish and Störrle [2007] use the term "pragmatics" (or implicature) for this information that can be obtained from the diagram, but was not encapsulated in the semantics or in the abstract model. The layout conventions of the secondary notation are usually developed by experts in order to convey more information than is actually contained in the abstract model. Such conventions are rarely formalized or codified which makes it hard or impossible to employ them in an automatic graph drawing algorithm. Examples in the software domain are the convention to draw a superclass above its subclasses in UML class diagrams (i.e., the inheritance relation is usually drawn bottom-up) or the placement of the nodes in UML activity diagrams so that the flow is from top to bottom or left to right.

The use of the secondary notation is, like good design, subject to personal style and individual skills [Petre, 1995]. Poor use of secondary notation which usually distinguishes diagrams drawn by novices from those drawn by experts makes representations more difficult to comprehend. Novices often fail to exploit the available layout cues (i.e., by dispersing related nodes) or even introduce mis-cues by for example using the Gestalt laws of symmetry or proximity arbitrarily so that the (experienced) reader draws incorrect conclusions. The secondary notation plays also an important role in the differences in strategies that are employed to read diagrams [Petre, 1995]. Readers have to identify and learn inspection strategies for diagrams which are processed and understood in parallel [Ware, 2004] and are therefore not amenable to the straight, serial reading style of text documents (see e.g. [Hornbæk and Frøkjær, 2003] for a discussion of reading patterns). The cues that can be used to navigate a diagram are usually part of the secondary notation. For example, the grouping or clustering of nodes can be exploited to reduce the search space and focus on that part of the diagram that contains the relevant information [Petre, 1995].

While the layout of a diagram is used by the secondary notation to add additional information, the geometric structure of the diagram also plays an important role in how the user interacts with a diagram over time. The user builds a strong "geographic" model [Bartram et al., 1995] in his

mind that is represented in the so-called “mental map” [Eades et al., 1991; Misue et al., 1995]. Users construct this mental map of the visualized information structure in order to effectively navigate and search for information [Huotari et al., 2004]. The mental map allows the user to reduce the space he has to search for a specific node or link. The empirical finding that visual recognition is a lot faster and more efficient than visual recall [Ware, 2004] indicates that the mental map is not an exact image of the diagram in the user’s head but rather a representation of its overall shape. Some formal measurements for the mental map and a discussion of the criteria that have to be fulfilled in order to preserve it are discussed in Section 5.3.

CHAPTER 3

Complexity of Graphical Models

Complexity is probably the single most important challenge the software engineering discipline faces today. The elements of a software system interact with each other in a nonlinear fashion, so that the complexity of the whole increases much more than linearly with the number of elements [Brooks, 1987]. The essential complexity of software systems stems largely from the complexity of the problem that has to be solved and the environment the system is embedded in [Simon, 1996]. As a result, the complexity of today's software systems presents an intellectual challenge without precedent in history [Dijkstra, 1989]. Models provide, as discussed in Chapter 2, through their inherent use of abstraction (or more precisely classification) a possibility to deal with this complexity. Furthermore, graphical models can make the intangible nature of software more tangible. However, as the complexity of software systems increase, so do their representations in graphical models. Or with the words of Edward R. Tufte [1990], "But, finally, the deepest reason for displays that portray complexity and intricacy is that the worlds we seek to understand are complex and intricate." Unfortunately, most current modeling notations and tools lack explicit complexity management mechanisms [Moody, 2006] so that they do not scale [Noik, 1994]. To make matters worse, graphical models of software systems are often used to communicate with non-technical people, who are far less used to deal with such complex (and additionally intangible) information. This often leads to a perceptual and cognitive overload [Moody, 2006].

The complexity of graphical models leads to two different but related classes of problems: First, the large number of elements and their mutual relations result in *information density problems*. And second, *spatial problems* occur because of the large space that is required to show all elements graphically. The following sections give an overview over various techniques that have been developed to cope with these problems. Section 3.1 describes techniques that try to solve

the problem on the presentation level. These techniques show all elements of the model all the time while providing special visualization or interaction functionalities to deal with large and complex models. In contrast, the techniques presented in Sections 3.2 and 3.3 employ additional abstractions that show different views on the modeled system either in different diagrams or at different levels of detail.

Most of the techniques presented in this chapter focus on the creation and use of graphical models with computer-based tools. While models on paper are much more tangible, the interactive nature of computer-based tools offers a lot more possibilities to deal with the arising information density and spatial problems. A desired view or perspective can always be created with the tool and then printed on paper to make it more tangible.

3.1 Flat Models

Most current modeling tools rely on flat or practically flat models and show all elements all the time in one view. Thus, the modeling tool is usually not much more than a simple drawing tool that supports and enforces only some very basic semantic concepts of the underlying modeling language. For example, the basic semantics of a node-link diagram are usually supported insofar as the lines between nodes that depict connections stay connected to the nodes when one of the nodes is moved. While the independence of the semantics of the supported notation restrains the range of possible techniques to cope with the size and complexity problems, it is at the same time the biggest strength of these tools as they are not restricted to any specific notation and can often be used for textual representations too. By their very nature, these techniques address the spatial problems (i.e., the presentation of large models on small screens) only and cannot do much about the information density problems.

3.1.1 Panning

One of the simplest possibilities to handle a model that is bigger than the size of the screen is panning, which is the "smooth, continuous movement of a viewing frame over a two-dimensional image of greater size" [Spence, 2007]. Only one part of the model is shown at a time and scrolling or paging provide (indirect) access to the rest. Two one-dimensional scrollbars are the usual way to pan. Each of these can be considered to provide a one dimensional overview of the document: the position of the scroll thumb (or knob) within the scroll trough (or gutter) portrays the location in the document while the length of the thumb shows the proportion of the document that is currently visible. The main problem with this scrolling is that most content is hidden from the current view [Spence, 2007]. This makes the orientation in the model or document difficult [Furnas and Bederson, 1995]. Different approaches have been developed to mitigate this problem by adding semantic information to the purely spatial information that is encoded in scrollbars (e.g., the "value bars" of Chimera [1992]). Furthermore, two independent

one-dimensional scrollbars result in some serious usability issues because a simple movement to a part that lies only slightly outside the currently visible part can require a series of alternating actions on the two scrollbars [Constantine and Lockwood, 1999, p. 205]. However, this problem has recently been mitigated by the advent of two-dimensional scroll wheels and the more direct panning possibilities of zoomable user interfaces (cf. Section 3.1.2).

3.1.2 Zooming

Besides panning, the other straightforward approach to deal with large models that do not fit on the screen is zooming (or linear scaling / projection / magnification). Zooming is the "smooth and continuously increasing magnification of a decreasing fraction (or vice versa) of a two-dimensional image under the constraint of a viewing frame of constant size" [Spence, 2007]. Like panning, it makes it possible to adjust the extract of the model that is currently visible, but provides a kind of exponential accelerator for moving around a very large space [Furnas and Bederson, 1995]. One big problem that arises with zooming graphical models of software systems is that the user can only enlarge one area of interest at a time (only one focal point). However, for program understanding it is often necessary to look at several disjoint parts of the software in detail at the same time (cf. Section 5.7 and [Storey et al., 1997a]). Zooming results in a temporal separation between contextual and detailed information [Cockburn et al., 2008] as detail and context can never be seen concurrently but only after each other.

Focus + Context Problem

The problem of integrating separated contextual and detailed information is known as the "focus + context" problem (the terms "detail-in-context" problem and "detail $\times$ scope" problem [Farrand, 1973] are also used). The loss of contextual information due to the absence of orienting features at high zoom levels leads to disorientation (or "desert fog" [Jul and Furnas, 1998]). In contrast to current modeling tools, the focus + context problem has been solved by the human visual system a long time ago. We can see different levels of detail because objects are seen at much lower resolution at the periphery of vision than in the center. The small resolution of current computer screens makes it impossible to use micro/macro designs [Tufte, 1990] which show both detail and global context in one representation and rely on our visual system to integrate detail and context. The resulting separation of focus and context makes the task of obtaining an overview notoriously difficult for electronic documents [Hornbæk and Frøkjær, 2003]. Mental maps (cf. Section 2.5) that play an important role in understanding and working with graphical models form easily and rapidly in environments where the viewer can see everything at once [Ware, 2004]. Animating zoom operations can help users to assimilate the relationship between different zoom states but can still not integrate local focus and global context. A visual exploration technique should ideally take advantage of the humans' natural visual pattern-recognition abilities to understand global relationships while simultaneously integrating this knowledge with local detail [Carpendale et al., 1997a]. This cannot be achieved with simple linear zooming.

Zoomable User Interfaces

The rapid zooming technique that is employed by zoomable user interfaces (ZUI) attempts to exploit the user's mental map and short term memory to provide context by only showing a small focus region like an ordinary zoom, but allowing the user to zoom-out very quickly to a low-detail context view and then zoom back in quickly to another focus region. Furnas and Bederson [1995] use the term "highly zoomable interfaces" for this class of techniques while Ware [2004] calls them "rapid zooming techniques". The main characteristics of zoomable user interfaces are that (i) the information objects are organized in space and scale and (ii) users interact directly with the information space, mainly through panning (which changes the area of the information space that is visible) and rapid zooming (which changes the scale at which the information space is viewed) [Hornbæk et al., 2002]. The first popular implementation of very fast zooming was the "Pad" system [Perlin and Fox, 1993] which additionally employs a "semantic zoom" so that the user sees different representations of an element when the magnification factor changes. For example, an element can appear as a rectangle if there is only little real-estate available and, as it grows, as a page of text. Elements can also disappear completely if they become too small. This technique of a logical or semantic zoom [Herot, 1980] generates a new representation when the zoom level changes instead of just globally scaling the whole representation.

A multitude of implementations of zoomable user interfaces have been proposed over time (e.g., Pad++ [Bederson and Hollan, 1994], Jazz [Bederson et al., 2000], Piccolo [Bederson et al., 2004b], ZVTM [Pietriga, 2005], MuSE [Furnas and Zhang, 1998]) and the technique has recently again gained a lot of attraction trough popular geographic information systems like Google Earth. An experimental evaluation of ZUI for software visualization of Summers et al. [2003] indicates that subjects perform better (i.e., faster and with fewer errors) by using rapid zooming techniques at understanding visual programs than with traditional zoom techniques. However, the direct interaction that is an essential part of ZUI's can become a problem if the model is editable as the user has to indicate whether he wants to move a node (or a set of nodes) or just the viewport.

Concurrent Control of Pan and Zoom

While traditional interfaces often combine panning and zooming, the two are usually controlled independently. This separation often results in a cascade of zoom and pan actions and thereby a cumbersome interaction. In order to overcome this problem, the approach of Bourgeois and Guiard [2002] assigns panning and zooming to two independent devices controlled by different hands to permit parallel control of zoom and pan. The "OrthoZoom Scroller" of Appert and Fekete [2006] is a one hand variant of this approach for one-dimensional (vertical) scrolling. It maps the vertical mouse movements to vertical panning and horizontal mouse movements to zooming. A different approach to connect zooming and panning is to automatically calculate one of the two from the user's manipulation of the other. The "depth modulated flying" (DMF) by Ware and Fleet [1997] automatically adapts the viewer's scroll velocity in response to his control of altitude (the zoom level). Tan et al. [2001] inverted the speed/zoom coupling with

the “speed-coupled flying with orbiting” in which the altitude (or zoom level) is automatically adjusted in response to the scroll velocity. The experiment of Cockburn and Savage [2003] shows that scrolling tasks are solved significantly faster with such an automatic zooming in both text document and map browsing tasks.

3.1.3 Overview + Detail

While zooming separates contextual and detailed information temporally, a small overview (or “map view”) combined with a large detailed window separates them spatially. The idea of combining a small panoramic overview (the “world view monitor”) and a large detailed screen goes back to the “Spatial Data Management System” [Donelson, 1978] developed in the 70’s at the MIT. The user interacts with the two views separately, although actions in one are usually immediately reflected in the other. This tight coupling between the overview and the detail view forbids an independent exploration of the two views. Recently, the boundary between such overviews and traditional scrollbars has been blurred by thumbnail document overviews that show small overviews of the pages of a document as part of the scrollbar.

A critical factor of this technique is the size of the overview window as it influences how much information can be seen in the overview and how easy it is to navigate in it. However, a large overview window might take too much screen real-estate from the detail window [Hornbæk et al., 2002]. This problem worsens with the size of the model, because the gap between the abstraction level in the overview and the detail view widens [Berner, 2002]. Some approaches allow the user to control the size of the overview to mitigate this trade-off.

A more fundamental problem is the integration of the two views that has to be done solely by the user. He has to figure out constantly where the small view fits in the big picture (even though the location and size of the detail window are usually indicated in the overview window) and what visible features correspond [Furnas, 1981] because an explicit connection between the detail and overview view is missing [Spence, 2007]. An experiment by Hornbæk et al. [2002] indeed showed that an overview slows users down. They suggest several explanations for the results: the overview might be visually distracting, switching between the detail and the overview window requires a mental effort and time to move the mouse or the overview window may have been to coarse (the overview window did not support fine-grained navigation). A second experiment [Hornbæk and Frøkjær, 2003] supported the first one insofar as subjects were slower at answering questions about an electronic (text) document with an overview + detail interface than with a linear interface. Interestingly, subjects scored significantly better in some tasks and strongly preferred the overview + detail interface. These findings were not supported by the experiment of Pietriga et al. [2007] which indicates that pan and zoom combined with an overview is more efficient than classical pan + zoom. Their work explicitly aims to eliminate cognitive skills and addressing motor and perceptual skills only (i.e., the subjects had to look for rectangles with rounded corners in a set of rectangles with square corners). However, exactly these cognitive aspects may cause interaction effects between the type of the task and the interface in real use.

3.1.4 Distortion

The main problem with both zooming and overview + detail is the separation between contextual and detailed information (temporal in the case of zooming and spatial for overview + detail) which places the demanding task of integrating the two solely on the user's shoulders. Distortion-oriented techniques aim to avoid this separation by displaying the focus within the context in a single continuous view. Presenting all regions in a single coherent display is meant to reduce the short term memory load associated with mentally integration distinct views and therefore potentially improve the user's ability to comprehend and manipulate the information. Since the screen (or more precisely its resolution) is usually not big enough to show the complete model, distortion-oriented techniques show the detailed information in a magnified focus region, while its surrounding regions present the global context at reduced magnification [Leung and Apperley, 1994]. In contrast to the conventional linear zoom (cf. Section 3.1.2), these techniques apply differential scale functions (i.e., nonlinear magnification) across the information surface, leading to intentional distortion. Thus, distortion-based tools provide space for the magnification of local detail by compressing the rest of the image [Carpendale et al., 1997a]. The loss of display area that results from the magnification of one or multiple parts of the image is compensated by the same amount of demagnification in other parts of the image. Otherwise, the area of the image would grow. A distorted view is generated by applying a mathematical function (the transformation function) to an undistorted image. This transformation function defines how the original image is mapped to the distorted view.

The application of distortion-oriented techniques to computer-based graphical data goes back to Farrand [1973]. However, the concept of distortion or deformation has been used over many centuries by cartographers in various map projections (see e.g., [Tufte, 1990, p. 12]). One example is Harry Beck's famous London Underground map. Beck realized that the exact geographic location and the distance between the stations can be abstracted away, even though this two seem to be fundamental properties of a map. Subsequently, he was able to freely distort the map preserving only the correct sequence of the stations which resulted in a much simpler and cleaner map. Leung and Apperley [1994] classify such distortion-oriented presentation techniques according to their magnification function in two distinct classes: techniques that have piecewise continuous magnification functions and techniques with continuous magnification functions. Two techniques with piecewise continuous magnification functions, the "Bifocal Display" and the "Perspective Wall", as well as one with a continuous magnification function, the "Graphical Fisheye Views", are presented in the following. We cover here only 2D techniques, a distortion-oriented technique for 3D can be found in [Carpendale et al., 1997a].

Bifocal Display

The Bifocal Display of Spence and Apperley [1982] represents the information space by employing the analogy of a sheet of paper which is folded to compress but still show the outer portions while leaving the center visible in full detail. The interaction with it can be described by the

metaphor of a paper stretched across four rollers, with two of them in the foreground giving a rectangular focal region and the other two rollers on either side giving receding side-plate displays. The user can now change the focus by sliding the paper in either direction. The Bifocal Display is relatively simple to implement and does provide spatial continuity between regions. However, it has the disadvantage of discontinuities of magnification at the boundaries between the detailed and the distorted view [Leung and Apperley, 1994]. Additionally, the information in the distorted regions of the Bifocal Display is often hard to decipher. That is not really an issue since the primary purpose of the distorted views is to let the user notice that there is some more information (i.e., awareness and identification). Text and other details that are unreadable can also be suppressed in the distorted regions which results in a kind of semantic zoom.

Perspective Wall

The Perspective Wall of Mackinlay et al. [1991] (and its successor the Document Lens [Robertson and Mackinlay, 1993]) upgrades the Bifocal Display with smooth transitions. Two side panels to the left and the right of the central panel show a distorted view of the out-of-focus regions by a direct demagnification that is proportional to their distance from the viewer. In contrast, the Bifocal Display has a constant demagnification factor. Therefore, the Bifocal Display is actually a special case of the Perspective Wall [Leung and Apperley, 1994].

Graphical Fisheye Views

The basic idea behind the Graphical Fisheye Views of Sarkar and Brown [1992] goes back to the Polyfocal Projection of Kadmon and Shlomi [1978] which uses the analogy of a map that is printed on a rubber sheet which is then pushed from behind at the focal point(s). The term fisheye stems from George W. Furnas' analogy to a very wide angle or fisheye lens (cf. Section 3.3.3 and Furnas [1986]). Sarkar and Brown proposed two different implementations, one based on a Cartesian coordinate system and the other on a polar system for the transformation[1]. Fig. 3.1 shows an example of a polar Fisheye View with the focus on the *CabinController* component.

Graphical Fisheye Views (or more generally presentation techniques with a continuous magnification factor) tend to distort the boundaries of the transformed image heavily because the image is transformed radially rather than independently in the x and y directions. This distortion increases with a bigger magnification factor [Leung and Apperley, 1994]. Users sometimes perceive the resulting Graphical Fisheye Views as too distorted and unnatural so that spatial comprehension can suffer a lot [Carpendale et al., 1997b]. Adding cues (e.g., simple parallel grid lines) has been proposed to enhance the user's spatial comprehension of the distorted space [Zanella et al., 2002]. A second problem are difficulties in target acquisition because the lens displaces items away from the actual screen position used to activate them. Thus, objects appear to move away

[1]The transformation was applied only to the nodes of a graph structure in order to avoid that straight lines and rectangles are transformed into curved lines and curvilinear rectangles respectively.

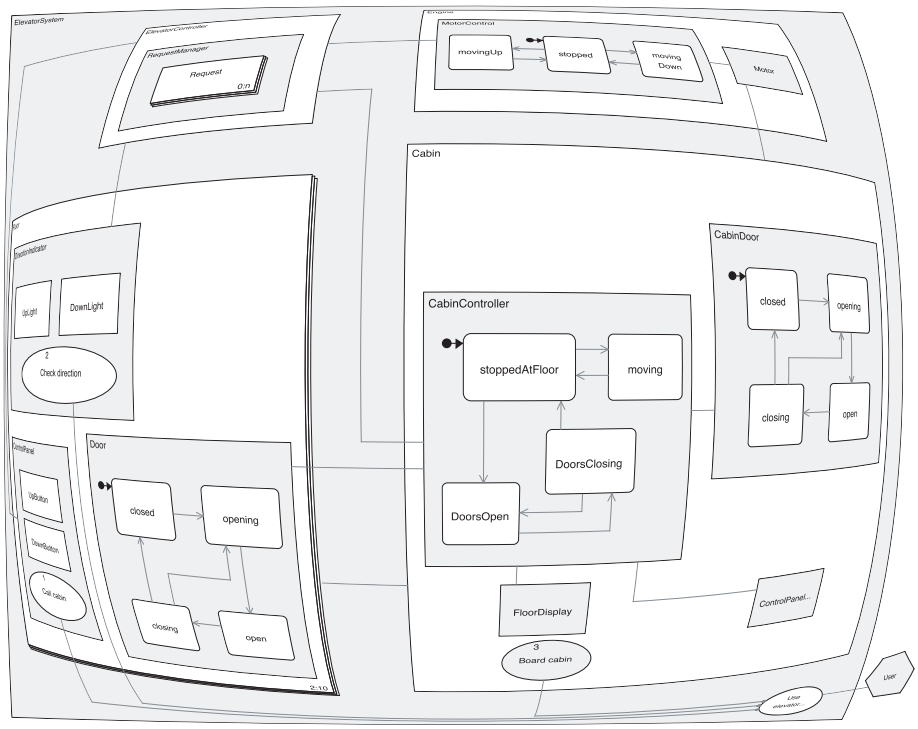


Figure 3.1: Graphical fisheye view of a simple software system model

as the focus approaches them (the so called “moving target” problem [Gutwin, 2002]). And finally, the interaction can become very cumbersome if Graphical Fisheye Views are applied on the model while it is edited because the interaction is not direct anymore but goes through the fisheye lens [Donskoy and Kaptelinin, 1997].

A number of experimental evaluations have been conducted during the last years in order to compare Fisheye Views with more traditional interfaces leading to contradictory results. The fisheye technique was significantly faster than either traditional zooming or panning for multi-point interactions in the experimental validation of Shoemaker and Gutwin [2007]. However, it remains unclear whether the improved performance was caused by the fisheye lens or the multiple focal points, because the control interface did not allow multiple focal points. Büring et al. [2006] compared a fisheye view and a traditional pan and zoom for searching scatterplot graphs on a mobile device, finding no significant difference in task times, but the subjects strongly preferred the fisheye interface (which actually contradicts [Baudisch et al., 2004; Donskoy and Kaptelinin, 1997; Gutwin and Fedak, 2004; Hornbæk and Frøkjær, 2003]). They suggest that the positive findings may be due to the nature of the task in the experiment which did not require relative spatial judgments. The task dependent results in the evaluation of fisheye views have also been shown in the study of Gutwin and Fedak [2004]. They compared a single-focus fisheye interface with a two level zoom and a pan interface for three tasks: image editing, web browsing and network monitoring. The fisheye interface best supported navigation, while the zooming

interface best supported monitoring and the traditional pan was the slowest interface for all three tasks. The results of Nekrasovski et al. [2006] were that pan and zoom was significantly faster than the focus + context technique. An additional overview did not impact task times, but the subjects favored an overview (which is supported by the findings of Hornbæk et al. [2002]). Furthermore, Donskoy and Kaptelinin [1997] showed that zooming outperforms fisheye views for tasks that involve identification and drag-and-drop manipulation of icons, which indicates that fisheye views can become problematic for interfaces requiring interactions which go beyond pure navigation (e.g., editing).

The idea of distorting the contextual information has also been applied to text (e.g., the "Fisheye Menus" of Bederson [2000]). The evaluation of a linear, fisheye and overview + detail interface for reading electronic text documents by Hornbæk and Frøkjær [2003] showed that the fisheye view encouraged faster reading, but at the expense of lower comprehension. The overview interface allowed the users to navigate the document rapidly, and although they spent more time reading, they scored better in comprehension tests (these results were supported by Baudisch et al. [2004]). Additionally, the linear interface for reading electronic text documents was inferior on most usability aspects compared to an overview+detail and fisheye interface. However, a following experiment with fisheye menus [Hornbæk and Hertzum, 2007] revealed that users make little or no use of the non-focal display region of fisheye menus and that performance with fisheye menus is inferior to traditional hierarchical menus which also points to the power and importance of a hierarchical structure (cf. Section 3.3).

3.2 Horizontal Abstraction

The techniques described so far address primarily the spatial problem (i.e., space limitation) that arises with large and complex graphical models. Dealing with the information density problem requires to take additional semantic information about the model into account in order to support additional abstractions. One such abstraction that is used frequently for graphical models of software systems is to provide views on the system from different perspectives [Kruchten, 1995]. These orthogonal views [Kim et al., 2000] on the description of a complete system are usually shown in separate diagrams. Typical views are a static or structural view that shows the structure of the system and a dynamic or a behavior view that describes how the system or its components perform over time. We use the term horizontal (or partitioning) abstraction [Berner, 2002] for this aspectual decomposition because the abstraction takes place within one hierarchical level. The complementary vertical abstraction of information on different hierarchical levels is discussed in Section 3.3.

The multi-diagram approach has been established as the de facto standard way to deal with the spatial and information density problem during the modeling of software systems by the UML (cf. Section 2.2.2). The smaller subdiagrams reduce the cognitive and perceptual overload, but introduce the problem of cognitive integration [Dori, 2002; Kim et al., 2000; Storey et al., 1997a]. The horizontal decomposition improves the understandability of individual aspects of

the system, but it may reduce the understanding of the system as a whole. As a consequence, the notation must provide explicit mechanisms for conceptual integration [Kim et al., 2000] to allow the reader the combine information from different diagrams into one coherent mental representation of the problem or system. This integration problem (or model multiplicity problem [Peleg and Dori, 2000]) comprises two subproblems [Kim et al., 2000]: First, during the *perceptual integration* the user has to establish the interdependence between the relevant system elements that have been dispersed across multiple diagrams. And second, the *conceptual integration* covers the generation and refinement of hypotheses about the system by combining the information inferred from the diagrams. The integration problem worsens with the fact that users often work on a small subset of the whole model, but on all aspects of this subset simultaneously. The perceptual and conceptual integration process can be supported by incorporating visual cues and contextual information in the diagrams as representation aids. One example of such aids are lines that connect the representations of the same element in different diagrams [Huotari et al., 2004].

A second, related problem are the frequent "context switches" [Donoho et al., 1988] that occur because the user is forced to constantly flip back and forth between two or more related diagrams. Henderson and Card [1986] use the term "thrashing" for this problem. Spatial contiguity can solve this problem by showing related diagrams simultaneously in the visual field [Moody, 2006]. This replaces a costly cognitive process (maintaining the information about the relations in working memory) with a much faster perceptual process (visual scanning). However, it is usually not possible to achieve such a visual contiguity on (small) computer screens. An additional more technical problem is maintaining the inter-diagram consistency to make sure that information concerning the same elements are consistent in different diagrams since the views employed by the different diagrams are usually not independent [Kruchten, 1995]. And finally, the main problem with a horizontal abstraction is that the individual diagrams can still grow very large without a sound hierarchical decomposition mechanism. Such hierarchical decompositions are the topic of the next section.

3.3 Vertical Abstraction

Probably the most successful way to deal with large and complex systems is a hierarchical decomposition. It breaks a complex system down into parts that are easier to conceive, understand and maintain. This divide and conquer approach is used in a wide variety of disciplines to deal with complexity (e.g., modularization in the system design and programming field [Parnas, 1972], segmentation in cartography or research in natural sciences in general [Simon, 1996]). A hierarchical decomposition that follows the principle of minimizing the coupling between parts while maximizing the cohesion of the individual parts allows to look at a system at different levels of detail while treating some parts as blackboxes (i.e., abstracting from their internals). We use the term vertical abstraction [Berner, 2002] for this form of abstraction that is based on different hierarchical levels. A hierarchical decomposition of a system's model into parts of manageable and understandable size has been one of the central pillars in the modeling of software

systems for years (cf. Section 2.2.1). But it had to take a back seat during the last years due to the lack of a clear decomposition construct in the UML. Only recently, with the large overhaul in version 2.0, has the OMG reclaimed the importance and power of hierarchical decompositions [Kobryn, 2004].

3.3.1 Graphical Representations of Hierarchical Decompositions

If only the hierarchical (i.e., parent-child) relationships are considered, a representation of a hierarchical decomposition as graph structure yields a tree (cf. Section 2.1.3). Thus, standard graphical notations for drawing trees such as graph trees or nested sets [Knuth, 1997, page 308] can be used to show hierarchical decompositions graphically.

Graph Tree

In a graph tree, all relationships, including hierarchical ones, are represented as edges and represented graphically as lines. The classical tree layout convention (see e.g. [Herman et al., 2000] for a survey of tree visualizations) which is for example employed in organization charts draws such a tree in a level fashion with the top level (i.e., root of the node) at the top. A decomposition can be represented graphically as tree by showing the part (component) as child of the whole (composite) node. The main drawback of the classical tree layout is that it becomes cluttered and unusable for large trees. Robertson et al. [1991] proposed "Cone Trees" as a solution to this problem by presenting the hierarchy in 3D to better exploit the available space. A hierarchy is laid out uniformly in three dimensions with a node at the top and its children placed evenly along the base of a cone. The additional dimension reduces the aspect ratio compared with a tree laid out in 2D. However, the interaction becomes more complex than in a 2D layout of the same tree. Another approach, the "Hyperbolic Tree Browser" of Lamping et al. [1995] uses hyperbolic geometry to visualize large trees. The hyperbolic tree browser stimulated the interest in using focus + context techniques for search tasks when a team using it outperformed five teams using competing browsers in a "browse-off" panel session at ACM CHI'97 [Mullet et al., 1997]. However, previous studies had failed to show significant advantages of the hyperbolic tree browser [Czerwinski and Larson, 1997; Lamping et al., 1995]. The studies of Pirolli et al. [2001, 2003] uncovered interaction effects between task and interface factors, with the hyperbolic tree performing well on tasks with strong scent, but performing poorly when scent is weak. Another way to deal with a large number of nodes in a graph tree is to directly employ vertical abstraction by expanding and collapsing branches of a tree dynamically [Plaisant et al., 2002].

Nested Sets

As an alternative to the classical tree layout, the nested set notation (or "inclusion tree layout convention" [Battista et al., 1994]) follows the concept of Venn or Euler diagrams by visually

representing the parent-child relationship by the child node being completely contained within the parent node. The notation is supported by the theory of pattern perception: The Gestalt principle of closure states that a closed contour tends to be seen as an object and that there is a very strong perceptual tendency to divide regions of space into "inside" and "outside" [Ware, 2004]. A region that is enclosed by a contour becomes a common region which is a much stronger organizing principle than for example proximity. Additionally, the semantics of the hierarchical decomposition become obvious as all children are removed automatically if the parent is removed. In contrast, the notation for composition and aggregation in UML class diagrams uses connecting lines instead of nested symbols. Additionally, the whole and its parts are shown on the same level. Thus, the Gestalt law of closure cannot be exploited which results in lengthy explanations and discussion of UML's aggregation and composition concepts. UML's composition and aggregation notation uses arbitrary symbols (cf. Section 2.2). In contrast to these arbitrary symbols, sensory symbols like graphically nested symbols derive their expressive power from their ability to use the perceptual processing power of the brain without learning [Ware, 2004]. The empirical evaluation of different representations of aggregation (or composition) by Irani et al. [2001] has revealed that the semantics that are inherent in the different kinds of relationships of real-world objects should also be applied to the abstract concepts in diagrams. Another advantage of the nested set notation is that some of the relationships (i.e., the parent-child relationship) between nodes are represented implicitly by the nested structure and do not have to be shown explicitly by a link which improves the readability of the diagram (by reducing the number of symbols). Furthermore, nodes can be summarized (i.e., vertically abstracted) by drawing the node rectangle without its contents.

One drawback of the nested set notation is that only one kind of hierarchical relationship can be represented by the nested symbols. If there are additional kinds of hierarchical relationship, they have to be depicted as lines between the nodes. Some approaches propose to show multiple kinds of hierarchical relationships directly by the nested set notation: The Higraph notation [Harel, 1988] represents both generalization and aggregation with nested boxes by dividing a node by dotted lines. And the Hicon approach [Sindre et al., 1993] shows several orthogonal hierarchical relations at the same time by a symbolic distinction of nodes (e.g., squares for aggregation and circles for generalization).

An additional problem is that the recognition of the hierarchical structure can become problematic with the nested set notation if only parts of the nodes are visible or connecting lines are confused with node boundaries. A possible solution is a simple shading of the nodes which alternates with the levels of the hierarchy (see Figure 3.4 on page 44). This exploits the Gestalt laws of figure-ground effects [Ware, 2004] that contribute to the perception of a figure because a node is now the figure on one and the ground on the next lower level. The shading of the hierarchy shifts some of the user's cognitive load to his perceptual system (i.e., substitutes seeing for reasoning). The way in which the visual system divides the world into regions is called segmentation [Ware, 2004]. According to the Gestalt laws of perception, the capacity of the working memory is limited by the number of chunks and not the size of each individual chunk [Moody, 2006]. Therefore, organizing the diagram elements in perceptual groups increases the

number of elements that can be shown in the diagram without overloading the working memory. Conversely, a missing structure or grouping of diagram elements requires each diagram element to be encoded as a separate chunk in working memory.

3.3.2 Explosive Zoom

Most current modeling tools allow the user to interact with hierarchically structured models represented in the nested set notation by what we call "explosive zooming" (or "multiple views" [Storey et al., 1997a]): when looking at the details of a particular node, the diagram with the details either replaces the previously displayed diagram or it is opened in a new window that supersedes the previously viewed one [Berner, 2002]. Thus, every level of the hierarchy consists of a set of separate diagrams which results in a cascading hierarchy of diagrams [Huotari et al., 2004]. In contrast to the traditional linear scale or zoom (cf. Section 3.1.2), the explosive zoom employs a structural zooming by exploiting the hierarchical structure of the presented information. The basic idea is to divide a single, large diagram along its decomposition structure into a set of smaller subdiagrams of manageable size. It is often recommended that this manageable size should be based on the limits of human information processing where the major constraint is the capacity of the working memory, which is believed to be "seven plus or minus two" [Miller, 1956].

Explosive zooming has the big drawback that the context of the zoomed node is lost: one cannot view the details of a node and its context in the same diagram. For human understanding of diagrams, this is bad because humans typically want to see their focus of interest in detail together with its surrounding context. The resulting integration problems are similar to those of traditional zooming (cf. Section 3.1.2), overview + detail interfaces (cf. Section 3.1.3) as well as the horizontal abstraction technique (cf. Section 3.2). Explosive zooming worsens these problems as it results in temporal *and* spatial separation. This is similar to hypertext systems with the same problem of not knowing one's location in the space and getting lost: clicking a hypertext link generates an entirely new screenful of information. The cognitive cost of such a simple interaction is that the entire information context has changed, and the new information may be presented using a different visual symbol set and different layout conventions. As a result, several seconds of cognitive reorientation may be required afterwards [Ware, 2004].

Diagrams are especially well suited to understand information (especially structural relationships) in parallel. The explosive zoom undoes this advantage as relationships over diagram boundaries are not visible. One possibility to mitigate this problem is the use of redundancy to achieve vertical integration, by employing foreign elements with a link back to their home diagrams whenever an element has a connection to an element in another diagram [Moody, 2006]. However, showing the same model element in different diagrams results in additional integration and consistency management problems. The emphasis of the UML on a horizontal abstraction together with its adoption of hierarchical decompositions in version 2.0 (the arbitrarily deeply nested hierarchical structures in composite structure diagrams, activity diagrams and state ma-

chine diagrams) has prompted many tool vendors to combine explosive zooming with the already employed scattering of different system aspects over multiple diagrams. This forces the user to mentally integrate horizontally (diagrams at the same level of the hierarchy) as well as vertically (diagrams at different levels of the hierarchy).

3.3.3 Suppression

While the horizontal and vertical abstraction as discussed so far use only the structure of the model to decide which elements are shown in the current view, the "suppression" [Spence, 2007] or "elision" [Ware, 2004] techniques take additionally the position of the beholder into account. The basic idea has been illustrated tellingly by Saul Steinberg in his famous New Yorker magazine cover "View of the World from the 9th Avenue", giving a Manhattanite's selective view of the world:

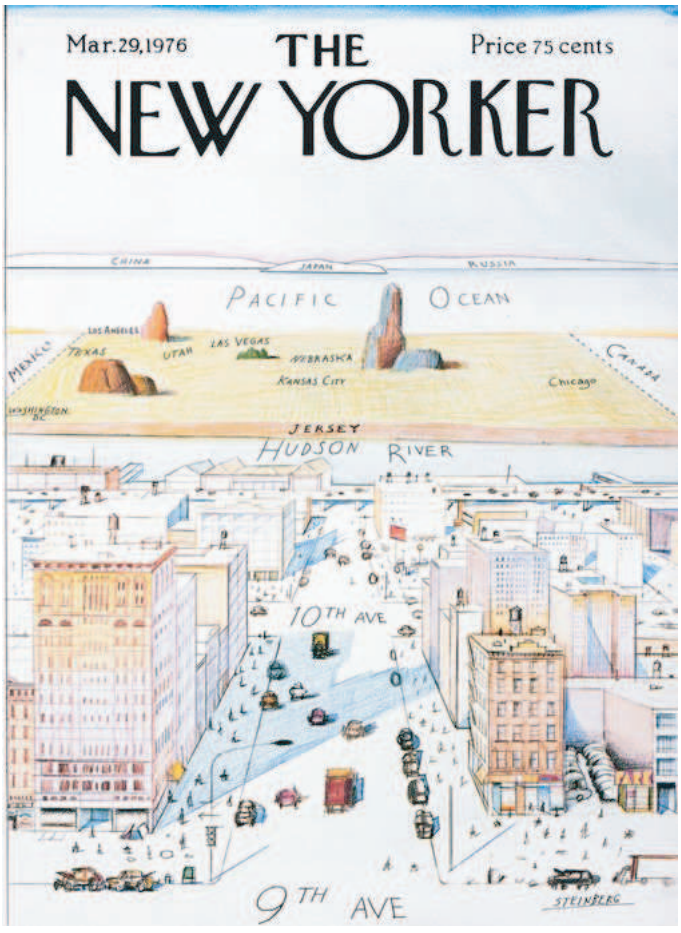


Figure 3.2: Saul Steinberg's "View of the World from the 9th Avenue"

Steinberg's illustration shows local detail (the 9th Avenue) together with global context. Different parts are shown at different levels of detail because they are, at the moment, of different importance for the beholder. More general, suppression techniques hide parts of the structure until they are needed. This is typically achieved by collapsing a large graphical structure into a single graphical object. This is analogous to the cognitive process of chunking, where small concepts or facts are cognitively grouped into larger chunks [Ware, 2004].

Instead of graphical or geometric distortion as done by the techniques presented in Section 3.1.4, suppression is used to create a detail-in-context (or focus + context) view. Distortion techniques show always all aspects of the image even if some of them are very compressed. They operate on the view and not the (hierarchical) structure of the model (i.e., the Graphical Fisheye Views can also be used on completely flat structures). In contrast, suppression techniques use structure-based abstraction instead of optical distortion so that the abstract structure (e.g., hierarchy) has to be taken into account.

Generalized Fisheye Views

One of the first application of suppression techniques to computers were in the 1980's George W. Furnas' "Generalized Fisheye Views" [Furnas, 1981, 1986]. Furnas proposed a semantic zoom which shows or hides an element in a structure depending on its "Degree of Interest" (DOI). The DOI is a function of the "a priori importance" of the element and its distance from the current focus. The user's DOI then determines which elements are displayed[2]. The result is a semantic zoom that suppresses information that lies away from the focal point. Furnas provided the theoretical foundations for focus + context interfaces by placing the concept of suppression on an intuitively attractive formal base. The analogy with the traditional fisheye view (or wide-angle lenses) is only on the conceptual level (i.e., by showing full detail in the center and less detail in the peripheral areas). The hierarchical structure of the information is used to decide which information should be shown and which information should be hidden. Furnas created systems for viewing and filtering structured programs, biological taxonomies and calendars.

The concept of the Degree of Interest can be illustrated by means of a simple tree such as the one in Fig. 3.3. Let's assume that the current focus lies on node F. The distance $D(n, f)$ between any node n in the tree and the current focus f can be calculated by simple measuring the distance (i.e., path length) between the two nodes. For the example of Fig. 3.3 this results in $D(A, F) = 2$, $D(B, F) = 1$ or $D(G, F) = 4$ etc. If we define 1 as an upper threshold for nodes that should be shown, only nodes B, F, I and J are shown in the current view (as indicated be the dashed line in Fig. 3.3). The concept of an "a priori importance" $API(n)$ of a node n allows to add a simple value to each node in the tree independently of the current focus or the position of the node in the tree. Let's assume that we set the values $API(A) = 10$, $API(B) = API(C) = API(D) = 9$, $API(E) = API(F) = API(G) = API(H) = 8$,

[2]The distortion techniques of Section 3.1.4 actually use the DOI implicitly to change the position and/or scale of the elements' representation.

$API(I) = API(J) = 7$ and $API(K) = API(L) = 6$ in the tree of Fig. 3.3 to indicate that the nodes at the top are more important than nodes at the bottom. The Degree of Interest $DOI(n, f)$ which is used to decide whether node n is visible if the focus is on node f is now calculated according to

$$DOI(n, f) = API(n) - D(n, f) \tag{3.1}$$

where n is the current node, f the current focus, $API(n)$ the a priori interest in node n and $D(n, f)$ the spatial or semantic distance between nodes n and f. For Fig. 3.3 this results in $DOI(A, F) = 8$, $DOI(K, F) = 4$ or $DOI(D, F) = 6$ and only the nodes within the dotted line are shown in the current fisheye view if we set the lower threshold to 5. The DOI function is an information suppression function and Furnas' approach therefore explicitly attacks the information density problem of graphical models. Thus, Generalized Fisheye Views create a kind of "information distortion" [Leung and Apperley, 1994] instead of a graphical or geometrical distortion. Probably the biggest drawback of Furnas' approach is that it supports only one focus at a time which is often not enough for the understanding of software structures. A hierarchical structure is not strictly necessary for the Generalized Fisheye Views, but Furnas' original publication introduces "the tree fisheye DOI" and the technique abstracts vertically (which is also the reason why it is introduced in this section).

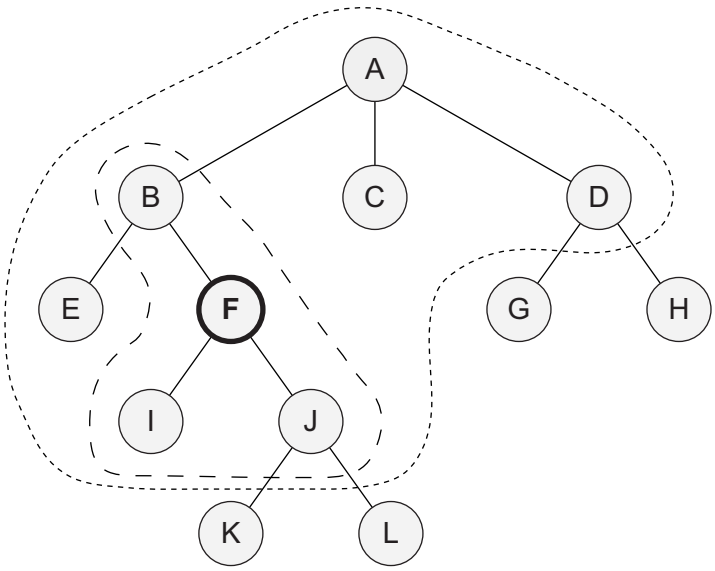


Figure 3.3: Generalized Fisheye View of a simple tree

Manual controls for collapsing and expanding program statement blocks and methods are now standard in most Integrated Development Environments (IDE). But automatic expansion and contraction using a DOI algorithm is still a research topic, as for example the Mylar tool [Kersten and Murphy, 2005]. Jakobsen and Hornbæk [2006] developed a fisheye source code viewer

that uses a DOI function to automatically display/suppress code blocks depending on the user's current focus. They found in an experimental comparison between the fisheye interface and a standard linear list view that the fisheye view not only resulted in significantly less time needed to fulfill a task but was also preferred to the linear listing. The concept has also been applied to other structures such as the "Table Lens" of Rao and Card [1994] which presents a compact overview of large tabular data, displaying all rows an columns simultaneously by showing values as small bars. Rows and columns can be selectively expanded to show the actual data values. Such expansions can be applied independently to rows and columns, allowing multiple focal points, while preserving the rectangular shape of cells. Bederson et al. [2004a] later applied the TableLens concept to a calendar in their "DateLens" approach .

3.3.4 Combined Distortion and Suppression

Furnas' DOI determines *what* information should be shown (the original formulation has no explicit control over the attributes that define the graphical layout), while much of the subsequent research addressed the issue of *how* the information is presented [Furnas, 2006]. The basic approach has been to apply an information suppression technique concurrently with a distortion-oriented technique to solve the spatial and information density problem at once [Carpendale et al., 1997a; Leung and Apperley, 1994]. Furnas' fisheye views can be applied to graphical models while avoiding the distortions that occur with nonlinear magnifications (cf. Section 3.1.4) by changing only the scale and/or position of an element but never its overall shape (i.e., a rectangle is still a rectangle after the distortion). If the model uses the nested set notation, fisheye views can be generated by collapsing subtrees of related nodes and edges into a single node to show different parts of the model at different levels of detail. Fig. 3.4 shows an example of such a fisheye view on a nested hierarchical graphical model.

The heart of such a fisheye view technique is its zoom algorithm which adapts the layout if a currently visible node is suppressed or a hidden node shown. By *zooming-in* and *zooming-out*, the user can control the level of detail of each node. Fig. 3.4a) shows an abstract view of a hierarchical model: only the three top-level nodes A, B and C are visible. The ellipsis after a node name indicate that the node has an inner structure which is hidden in the current view. By successively zooming-in nodes B and D, we get a view, shown in Fig. 3.4b), which shows the details of the model in a focal point (node D) together with the global structure of the model. Conversely, by zooming-out nodes in an expanded model we get a more abstract view. This concept of zooming-in and -out is not new, it has for example already been proposed by Harel for his statecharts [Harel, 1987]. Robertson et al. [1991] call them "gardening operations" because they prune and grow the view of a tree. The terms that are used to describe the resulting views vary widely among different authors, besides the already used "fisheye view" and "focus + context view" these are "detail-in-context view" [Carpendale et al., 1997a], "multiscale view" (i.e., different sections of the information are displayed at different scales) [Furnas and Bederson, 1995] or "distortion view" [Leung and Apperley, 1994].

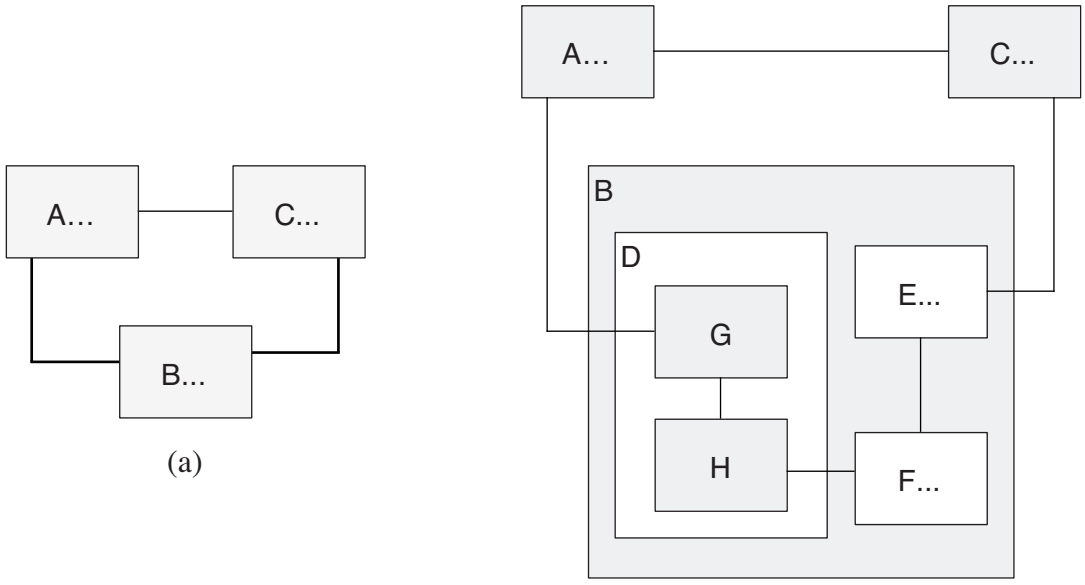


Figure 3.4: Fisheye zooming

CHAPTER 4

ADORA

Most current modeling tools rely on flat or practically flat models (i.e., they show all elements in one view) and support only panning and scaling for navigation (cf. Section 3.1). Tools that support a navigation trough the hierarchical decomposition of a model visualize only one level of the hierarchy using an explosive zoom (cf. Section 3.3.2). Given this state of tools, it is not surprisingly that most modeling languages rely on the principle of loosely couple multi-diagram models as the primary means for separating concerns and decomposing large models into manageable parts. Since tools do not really support hierarchical and aspectual decompositions of large models, decomposition techniques can be provided on the language level only. The basic idea of the ADORA[1] approach is to reverse these principles by using an integrated, inherently hierarchical model instead of a loose collection of diagrams and a tool that generates abstractions and diagrams of manageable size. This chapter gives a short overview over the language (Section 4.1) and tool concepts (Section 4.2).

4.1 The ADORA Language

ADORA has been developed at the Requirements Engineering Research Group of the University of Zurich as a language for modeling requirements and, in later stages of the development process, architectural design specifications on the logical level [Berner, 2002; Glinz et al., 2002; Joos, 1999; Meier, 2009; Seybold, 2006; Xia, 2005]. The driving force behind its development is to think beyond currently established modeling paradigms and languages and to mitigate some

[1] ADORA is an acronym for **A**nalysis and **D**escription **O**f **R**equirements and **A**rchitectures.

of the current modeling problems and anomalies. The following sections give a short overview over the most important language concepts of ADORA and compares them with the current state of the art (which is basically UML). A more detailed description of the current state of ADORA can be found in [Meier, 2009].

4.1.1 Abstract Objects Instead of Classes

Instead of types or classes which are the basis for most object-oriented modeling languages, ADORA is based on abstract objects. Abstract objects are prototypical instances of types or classes. They stand for the concrete instances and their location in the object structure of a system. However, since they do not represent objects at runtime, there are no concrete values stored in their attributes. Object models are more powerful than class models in expressing composition relationships [Joos, 1999]: With class models it is not possible to model the context in which an instance of a class is used. Furthermore, class models fail if more than one object of the same class has to be modeled [Glinz, 2000] and are hard to decompose [Glinz et al., 2002].

Another problem with class models are the relationships they support. Riel [1996, p. 53] distinguishes between the following categories of relationships that occur in the object-oriented paradigm: (i) uses relationships, (ii) containment relationships, (iii) inheritance relationships and (iv) association relationships. Out of these only the inheritance relationship is class-based while the other three are object-based. The concept of class- and object-based relationships denotes whether or not all objects of a class have to obey the relationship. Because UML (cf. Section 2.2.2) puts the emphasis on inheritance[2], the class model is its main cornerstone even though the other types of relationships do not really fit into it (because they represent relations between objects and not classes). UML 2.0 has shifted the focus a little bit from classes to components. However, this has let to a lot of syntactic and semantic overlap between the concept of a class and a component [Kobryn, 2004].

4.1.2 Hierarchical Decomposition

As discussed in Section 3.3, a hierarchical decomposition is probably the best way to handle big models and deal with their complexity. In ADORA, abstract objects are recursively decomposed into objects. Thus, the object concept goes over all levels of the hierarchy and the only thing that changes is the degree of abstractness: objects on lower levels model small parts in detail, while objects on higher levels represent larger parts or the whole system on an abstract level. This makes is possible to use the same graphical notation in summary and detail diagrams. Other approaches in which the concepts differ among hierarchical levels have to use different notations [Moody, 2006]. Using a fisheye zoom concept on this uniform structure of nested objects results

[2] UML's strong emphasis on inheritance has also led to conceptual critic. For example, the fact that inheritance breaks encapsulation has made Gamma et al. [1995, p. 18f] to recommend favoring object composition over class inheritance.

in a full vertical model abstraction mechanism. The user can decide on the level of detail in any part of the model without any restrictions on the number of levels. While previous versions of UML relied almost only on horizontal model abstractions, classes and components can finally be decomposed in UML 2.0 [Kobryn, 2004]. This hierarchical decomposition of structure and behavior includes black- and white-box views on classes and components which makes is possible to apply such a zoom approach to UML as well.

4.1.3 Integrated Model

Instead of a hierarchical decomposition, most existing object-oriented modeling languages, with UML as the most prominent example, use a set of more or less loosely coupled diagrams of different types to cope with the complexity of the modeled system. While this approach facilitates a clear separation of concerns, a collection of different models hinders integration and abstraction tasks. The lack of integration on the language level results in more redundancy within the language [Joos, 1999]. Enforcing the horizontal consistency between different models demands additional constraints. Furthermore, the whole integration task is simply left to the user who has to integrate the separate models mentally (cf. Section 3.2).

In order to mitigate this problems, ADORA integrates all modeling aspects (structure, data, behavior, user interaction, etc.) in one single coherent model. This facilitates a strong notation of consistency and provides the basis for the development of powerful consistency checking mechanisms. Additionally, it releases the user, at least partially, from the demanding integration task. UML supports since its significant revision in version 2.0 the cross integration of structure and behavior. However, other aspects such as the user interaction (expressed with use cases in UML) are still not well integrated with the rest of the language [Kobryn, 2004].

4.1.4 View Concept

While the use of an integrated model facilitates better model understandability, the user is drawn in a flood of information if every element of the model is shown all the time. As most of these elements are not in the reader's (current) focus of interest, the cognitive overload can be reduced by showing only the model elements which are of interest at the moment. ADORA supports this by two visual abstraction mechanisms: (i) the model is decomposed hierarchically (as explained Section 4.1.2), thus allowing the user to select the vertical abstraction level. And (ii) ADORA employs a view concept that is based on different aspects (structure, data, behavior, user interaction, etc.) of the system. Thus the complete model is basically an abstract one that is virtually never shown in a diagram. The concrete diagrams illustrate certain aspects of certain parts of the model in their hierarchical context. This horizontal model abstraction mechanism is achieved by using the basic fisheye zoom algorithm to dynamically generate views by toggling the visibility of one or more model elements (cf. Section 8). Such a view is in the mathematical sense a projection of the model. ADORA supports six different views:

- The **base view** shows the static structure of the system. Its hierarchy of abstract objects forms the basic structure of an ADORA model. Objects are represented by rectangles using the nested box notation (cf. Section 3.3.1). This allows the user to represent the model at any level of abstraction and thus to fully exploit the power of the hierarchical decomposition. As the base view represents the basic structure of an ADORA model, it is shown all the time albeit at different levels of detail.

- The **structural view** augments the base view with information about relationships between objects. The ADORA language offers associations to model structural relationships or a communication channel between two objects. Because associations must reflect the hierarchical structure of the model, ADORA uses the concept of abstract associations (cf. Section 11.1) which represent one or multiple associations on a lower level of abstraction as soon as one of the participating nodes is not visible anymore.

- The abstract behavior of objects is represented by the means of statecharts [Harel, 1987] in the **behavior view**. States are embedded in the base view and represented by rounded rectangles which can be nested. State transitions are shown as arrows connecting the states.

- The **user view** is used to model the system from the viewpoint of the user(s). The interaction between the system and objects in its environment is modeled by scenariocharts [Xia, 2005]. Scenarios are the ADORA equivalent to UML's use cases. They are visualized as ellipses and can be decomposed into sub-scenarios forming a Jackson-like [Jackson, 1975] scenario tree. These scenariocharts are integrated into the base view so that a scenario is placed inside the node that represents the object the environment object interacts with in this (sub)scenario.

- The actors in the context of the system are modeled in the **context view**. Actors are represented graphically as hexagons and interact through scenarios with the system. Depending on the level of abstraction the system is currently shown with, the context view yields a context diagram showing the system as black box together with its external actors.

- ADORA's **functional view** is used to define additional properties (e.g., attributes or operations) of objects. The functional view is not shown together with the other views but displayed separately because it consists mainly of text.

The view concept of ADORA is similar to the floor plan that is the base view for the civil engineering field which is augmented with additional layers (with their own notation) for electricity, heating and other aspects of a building.

4.1.5 Variable Degree of Formality for a Controlled Evolution

ADORA allows its users to adapt the degree of formality in the model to the difficulty and risk of the problem at hand. All language elements support a variable degree of formality so that

they can be represented informally, semi-formally (which is a mixture of formal and informal elements) or formally. For example, the specification of the behavior of an object can range from informal natural language to fully-fledged statecharts. Most of the time a model consists of both formal and informal constructs while it evolves towards a complete and formal specification.

Such an evolutionary requirements process is explicitly supported by ADORA through a semi-formal simulation of a model [Seybold, 2006]. Any requirements modeling language should support model evolution because requirements are not complete and clear from the beginning but have to be elicited and refined over time. Model elements can be marked as intentionally incomplete in ADORA so that they can be distinguished from elements that are incomplete by mistake (i.e., unintentionally). These incomplete models can then be simulated in order to uncover errors end fill the gaps.

4.2 The ADORA Tool

The ADORA tool does not only facilitate the creation of an ADORA model but is, with its navigation capabilities, an integral part of the approach. Thus an implementation of the concepts in a tool is not only desirable but essential. Furthermore, an implementation of the concepts presented in this thesis is necessary to verify their feasibility, identify and correct weaknesses and to validate them. The fisheye zoom and line routing concepts have in fact not been developed in a purely theoretical manner but evolved from an iterative mixture of conceptual development and implementation.

The current version of the ADORA tool prototype is based on the Eclipse[3] platform, while its predecessor was a pure Java implementation. One of the most powerful features of the Eclipse platform is its plug-in architecture which facilitates the extension of the platform in a clear way. One such extension is the Graphical Editing Framework (GEF)[4] which provides a Model-View-Controller framework for the implementation of graphical model editors. The Graphical Editing Framework employs basic support for recurring features of graphical editors (e.g., basic undo or copy & paste support) which reduces the implementation and especially maintenance load while it improves the usability of the tool significantly.

A detailed discussion of the ADORA tool can be found in Chapter 15 which shows how the concepts presented in the following chapters have been implemented in the tool.

[3] http://www.eclipse.org
[4] http://www.eclipse.org/gef

Part II

Fisheye Zooming

CHAPTER 5

Desired Properties of a Zoom Algorithm

A zoom algorithm has the responsibility to adjust the layout of a diagram (i.e., the position and size of the nodes) if one or multiple nodes are suppressed to reduce the overall complexity of the model or shown to see the details within one part of the model. A set of criteria (or desired properties) is needed to assess how well different techniques are suited for this task. This chapter lists the properties that a zoom algorithm should have if it is used for visualizing and editing hierarchical models. The properties are derived from the literature on focus + context visualization (cf. Chapter 6) and our own previous experience with the ADORA tool (cf. Chapter 4.2). The presented properties differ from the aesthetic criteria that are used by the automatic graph drawing algorithms (cf. Sections 2.3.1 and 2.4) because they are used for layout adjustment operations and not the creation of an initial layout.

5.1 Compact Layout

Reducing the complexity and size of diagrams is the main motivation for the work presented in this thesis. The complexity is reduced by suppressing model elements from the diagram. In case of the fisheye zoom the internals of a node are hidden once the node has been zoomed-out. The white space that becomes available at the position of hidden nodes must be removed by the zoom algorithm to reduce the size of the whole diagram. The sibling nodes of the currently suppressed nodes are moved which results in a global distortion of the diagram. This has to be done recursively if a hierarchical model is represented as a set of nested boxes. The zoom algorithm must only distort the position of the nodes and not their shape because of the problems with

the readability of text and the recognition of links in classical distortion techniques (cf. Section 3.1.4). Producing a compact layout is the rationale for developing a zoom algorithm that reduces the size of the diagram if a node is zoomed-out [Misue et al., 1995]. However, the demand for a compact layout can also become problematic if some elements are suddenly too close together or even overlap with other elements such as links or labels (cf. Chapters 11 and 14). This trade-off between a compact layout on one side and an easily readable and understandable diagram on the other may result in the need for different compacting techniques for different tasks.

Fig. 5.1 illustrates the basic idea of a compact layout: Nodes G and H are suppressed from Fig. 5.1a) to get a more abstract view of node D in Fig. 5.1b). Node D shrinks because none of its children is shown anymore. Nodes E and F are moved closer together so that the newly available white space is not wasted. This reduces the space that is required by node A to show its children which lets node A shrink. The smaller size of A results in more white space which can be used by A's siblings B and C. The whole diagram then becomes smaller because nodes B and C are moved closer to node A.

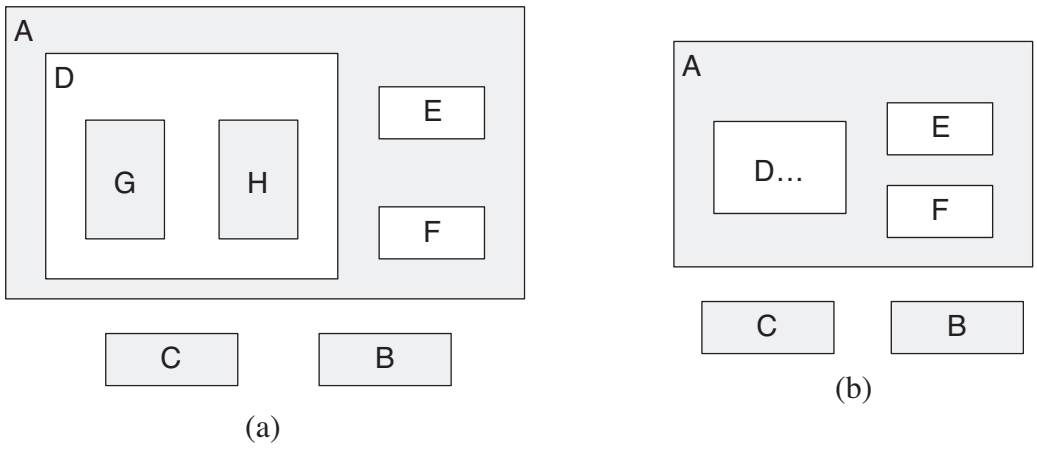


Figure 5.1: Produce a compact layout

5.2 Disjoint Nodes

The zoom algorithm is restricted by a set of constraints while it moves nodes to produce a more compact layout. The most important constraint is that nodes on the same level of the hierarchy must not overlap [Bartram et al., 1995; Misue et al., 1995]. Overlapping nodes reduce the readability of the diagram because some informations may be hidden behind nodes. Additionally, it becomes difficult to see the hierarchical relationships between nodes because these relations are represented as nested boxes. A node that is located inside another node just because the two nodes overlaps compromises this notation.

The property of disjoint nodes must hold after any zoom operation regardless of whether a node is zoomed-in or out. For the example of Fig. 5.1 this means that nodes D, E and F (which are on the second level of the hierarchy) as well as nodes A, B and C (which are on the first level) must not overlap in Fig. 5.1b) if node D is zoomed-out. The same must hold for the inverse operation (i.e., the resulting layout of Fig. 5.1a) if node D in Fig. 5.1b) is zoomed-in). If the property of a compact layout is the rationale for a zoom algorithm in the case of zoom-out operations, then the property of having disjoint nodes is the rationale for zoom-in operations: Zooming-in expands nodes so that sibling nodes have to be moved away to avoid overlaps. There are essentially two different ways to deal with the property of disjoint nodes: Either the zoom algorithm itself guarantees an overlapping free layout or overlaps are detected and resolved explicitly after the nodes have been moved by the zoom algorithm.

5.3 Preserve the Mental Map

To maintain the understandability of a model, it is vital that the new layout resembles the original layout as far as possible after an adjustment of the layout (such as a zoom operation) [Bartram et al., 1995; Carpendale et al., 1997a; Leung and Apperley, 1994; Misue et al., 1995; Noik, 1994; Sindre et al., 1993; Storey and Müller, 1995]. This is important because the modeler builds a mental map which consists of the positioning, the size and other user-defined visual properties of the model (cf. Section 2.5). A new layout that differs too much from the current layout results in a spatial disorientation of the user. The layout characteristics that can be used as metrics to formalize the mental map and that should be maintained in an adjusted layout to preserve the user's mental map depend on the type of the layout and the application. Thus, they vary widely among different authors:

- The relative direction between nodes [Sindre et al., 1993] or the relative location of nodes [Bartram et al., 1995] should be maintained. Eades et al. [1991] and Misue et al. [1995] use the term **orthogonal ordering** of nodes for this characteristic and try to formalize it by stating that it is preserved if the horizontal and vertical ordering of points is preserved. The position of the center of the node v is given by its coordinates (x_v, y_v). The adjusted coordinates (x'_v, y'_v) and (x'_u, y'_u) of two nodes v and u have the same orthogonal ordering as the coordinates in the original layout if

  - $x_u < x_v$ if and only if $x'_u < x'_v$ and
  - $y_u < y_v$ if and only if $y'_u < y'_v$ and
  - $x_u = x_v$ if and only if $x'_u = x'_v$ and
  - $y_u = y_v$ if and only if $y'_u = y'_v$.

  Two layouts have the same orthogonal ordering if these relations hold for each $u, v \in V$, where V is the set of all nodes. Bridgeman and Tamassia [2000] use the relative angle

between pairs of points to measure differences in the orthogonal ordering. Their qualitative experiment which compared the ordering by similarity of drawings of slightly modified graphs by a metric to the one of a human, revealed that the orthogonal ordering metric performed strikingly better than most of the other examined metrics.

- The relative distance between nodes [Sindre et al., 1993] or the **proximity relations** [Misue et al., 1995] are maintained if nodes that were close in the original layout are still close in the adjusted layout. The motivation behind the preservation of proximity relations[1] is to capture clusters or visual chunks that a user may use as landmarks for the orientation.

  There are many ways to mathematically capture the intuition that items that are close together in the original layout should stay close together in the adjusted layout. Bridgeman and Tamassia [2000] distinguish between the "nearest neighbor within" and "nearest neighbor between" as a measurement to compare how near a pair of points is in the original and the adjusted layout. The nearest neighbor within metric is based on the reasoning that the closest point of any point in the original layout should also be the closest point in the adjusted layout. The nearest neighbor between measures whether a specific point in the adjusted layout is nearer to its original position than any other point if the original and adjusted layouts are aligned. Misue et al. [1995] propose the use of a proximity (or neighborhood) graph whose nodes are the points the proximity should be measured for and whose edges are defined by some kind of proximity relations between these points. They suggest a few possible proximity graphs: the convex hull and the minimum spanning tree as simple examples, the Delaunay triangulation as a very strong notation of proximity or the sphere of influence graph as a more sophisticated concept.

- Different **distance** metrics have been proposed to measure the distance from the original to the adjusted layout. Lyons et al. [1998] use the Euclidean distance as a simple metric to measure the average distance moved by each point from its position in the original layout to its new position in the adjusted layout. In order to measure how much the points in the original and adjusted layout move relative to each other, Bridgeman and Tamassia [2000] propose the "relative distance" which measures the average change in distance between each pair of points in the original and adjusted layout.

- The relative **size of nodes** is mentioned by Sindre et al. [1993] and Storey and Müller [1995]. However, none of them gives an explanation of what exactly should be preserved.

- Storey and Müller [1995] argue that some characteristics of the **links** that connect the nodes should be maintained too. They suggest the straightness of links and the orthogonality of links parallel to the X and Y axes as important characteristics that should be maintained to preserve the mental map. The "shape" metric of Bridgeman and Tamassia [2000] is also motivated by the reasoning that the routing of links may have an important effect on the overall look of a graph and therefore on the user's mental map. They define the shape of a

[1] Bridgeman and Tamassia [2000] additionally distinguish between proximity relations and metrics that are based on partitionings other than proximity.

link as the sequence of directions (north, south, east and west) traveled while following the link (for non-orthogonal links the most prominent direction is used). The "shape string" is then composed by writing the link's shape as a string of N, S, E and W characters. For each link the edit distance (i.e., minimum number of insertions, deletions or replacements of characters needed to transform one string into the other) between the shape string of the original and the adjusted layout can then be computed to measure the similarity. A more detailed discussion of the impact of links on the mental map can be found in Section 11.3.5.

- The **topology** metric is motivated by the idea that the order of links around a node plays an important part in preserving the mental map. According to Misue et al. [1995] it is preserved if the original and the adjusted layout have the same dual graph. The dual graph is defined as follows: The layout of a graph consists of nodes that represent the vertices and links representing the edges. These links divide the region of the plane not occupied by nodes into subregions which are called faces. A face can be identified by the list of links and nodes encountered in a clockwise traversal of the boundary of the face. The dual graph has these faces as vertices and a link between two vertices if they share a common boundary in the layout. Fig. 5.2b) shows the dual graph for the layout of Fig. 5.2a). Node 1 in the dual graph represents the face $AbBcCgFdeDfEfDa$ (the so called outside face), node 2 face $AdeDa$, node 3 face $CgFde$ and node 4 face $AbBcCed$.

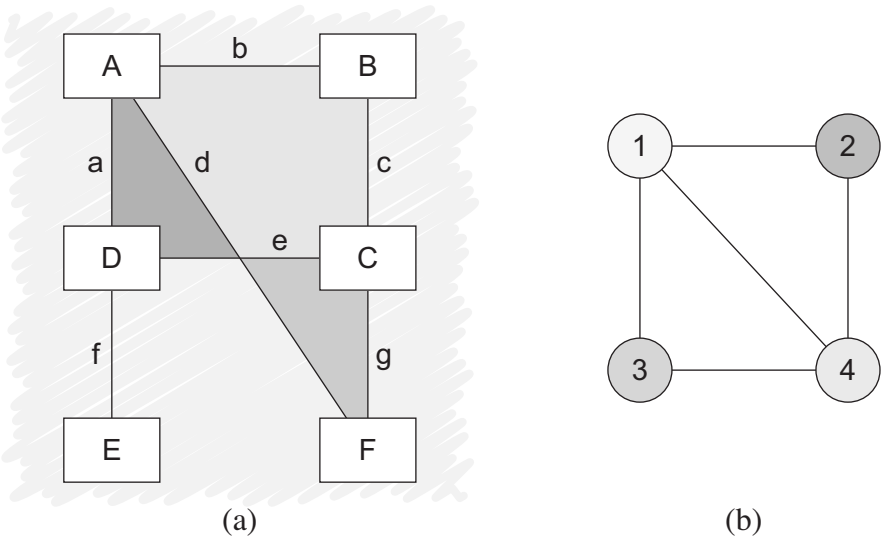


Figure 5.2: Dual Graph [Misue et al., 1995]

5.4 Layout Stability

Of specific importance in order to preserve the user's mental map is the stability[2] of the layout: If a sequence of zoom operations is followed by a sequence of inverse operations, the result should be the original layout. A hard lesson that we learned from using our ADORA tool (cf. Chapter 4.2) is that, when a sequence of zoom operations is reversed, it does not suffice to produce a layout similar to the original one in terms of orthogonal ordering, proximity and topology. The resulting layout must be identical to the initial layout because of two reasons. First, the user should be able to recognize a layout that he has seen before so that the current view of the model fits into his mental model. And second, the user has usually spend a lot of time in manually arranging the layout and has encoded a lot of additional information in its secondary notation (cf. Section 2.5). Therefore, the layout of a specific view of the hierarchy must always be the same irrespective of the sequence of zoom operations that led to it. We can define the stability property in a more formal way by means of Fig. 5.3:

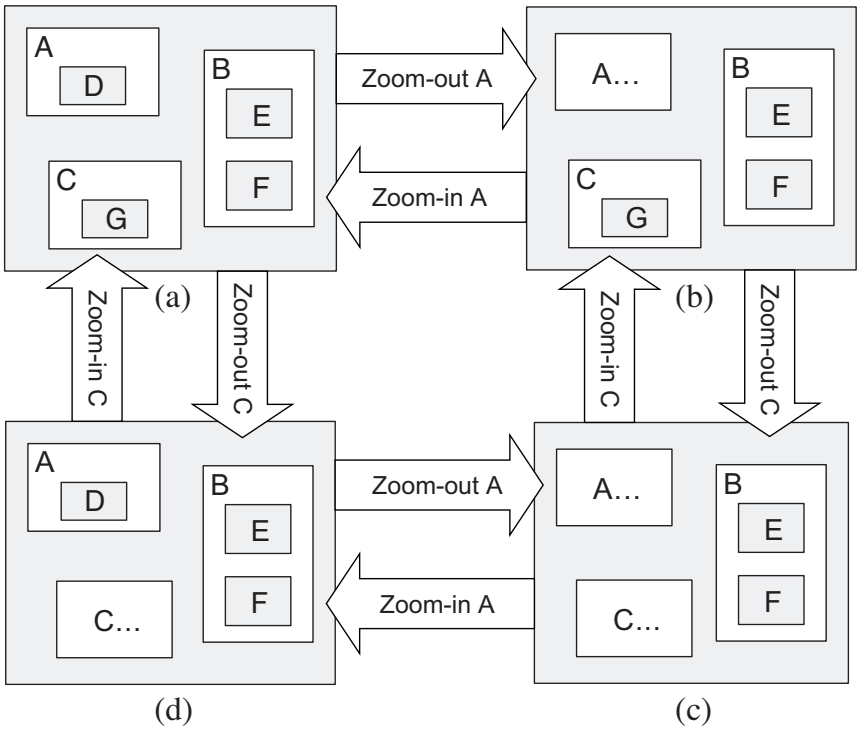


Figure 5.3: Layout stability

We first define the *zoom state* $zs(n)$ of a node n which can either be in or out. The zoom states for the nodes in Fig. 5.3a) are: $zs(A) = in, zs(B) = in, zs(C) = in, zs(D) = in, zs(E) = in,$

[2]Note that North [1996] uses the term "incrementally stable" to describe the preservation of the mental map that was discussed in the previous section.

$zs(F) = in$ and $zs(G) = in$. Note that the zoom state of notes that do not have any children and cannot switch their state is always set to in. The zoom state of node A changes in Fig. 5.3b) to $zs(A) = out$ because node A is zoomed-out while the zoom states of the other nodes remain unchanged. Furthermore, we define the zoom state of the model $mzs(\Theta)$ as the set containing the zoom states of all nodes, so that the model zoom state $mzs(\Theta_a) = \{zs(A) = in, zs(B) = in, zs(C) = in, zs(D) = in, zs(E) = in, zs(F) = in, zs(G) = in\}$ is shown in Fig. 5.3a). Nodes whose parent is in the zoom state *out* can be omitted from the state definition, so that the model zoom state of Fig. 5.3c) is defined as $mzs(\Theta_c) = \{zs(A) = out, zs(B) = in, zs(C) = out, zs(E) = in, zs(F) = in\}$. The number of model zoom states (i.e., the model zoom state space) can become quite large because it contains n^2 states, where n is the number of nodes that have children and can therefore be zoomed.

A model zoom state $mzs(\Theta)$ describes a specific view on the hierarchy of the model (i.e., which nodes are currently visible because their parent node is zoomed-in or suppressed because their parent is zoomed-out). The layout stability characteristic states now that there exists exactly one layout for each model zoom state $mzs(\Theta)$. Any sequence of zoom operations that results in a specific model zoom state must also result in this specific layout. For example, zooming-out node A in Fig. 5.3a) results in the layout of Fig. 5.3b). Zooming-out node C and then zooming-in A produces the layouts of Fig. 5.3c) and Fig. 5.3d) respectively. If the next zoom operation is a zoom-in of node C then the layout must be identical to the initial layout of Fig. 5.3a). This property must hold even though the inverse zoom operations have not been performed in strictly reverse order. This means that the zoom operations must be commutative (i.e., the order in which they are performed must not have any influence on the final layout).

5.5 Permit Editing Operations

Graphical models are manually and incrementally built by a modeler, so that the models are not static but change and evolve over time. The possibility to directly manipulate the model is one of the biggest advantages of computer-based modeling tools over paper-based modeling [Scaife and Rogers, 1996]. The technique that is used for their visualization must support manual modifications of the model. A user may re-arrange the layout of a model or he may change the model by adding or removing elements. This is not the case for "pure" visualization or presentation tools that are used in the reengineering field and show an existing structure (e.g., the program code) which is not modified anymore. Thus, we can distinguish between visualization tools that show a static existing structure and graphical editors that are used to dynamically and incrementally build the structure. Since we want to use the zoom algorithm for graphical editors, it must permit editing operations (add, remove, move and resize nodes) and should try to preserve the modified layout as far as possible in subsequent zoom operations. The user must be able to edit the diagram regardless of the current view (i.e., the current zoom state) of the model because he is almost always working on an abstracted view (i.e., with some of the nodes hidden) and almost never on the fully expanded model.

As the editing of models that use the nested box notation employs specific problems (cf. Chapter 9), the zoom algorithm should support automatic layout expansion when an inserted node is larger than the empty space at the insertion point as well as automatic layout contraction when a node is removed. The existing layout should be maintained as far as possible in case of this automatic expansion and contraction in order to maintain the user's mental map (cf. Section 5.3). However, this is only possible to some extend because the stability property (cf. Section 5.4) may not hold anymore if the model has been edited between zoom operations. The need to support editing operations renders approaches which simply store the layout of all zoom states in order to reconstruct the layout impossible (because changing one layout may change all stored layouts).

5.6 Runtime

Zooming is part of an interactive viewing, navigating and editing process. Hence, the zooming operations have to be performed with no remarkable delay even for big models [Bartram et al., 1995; Leung and Apperley, 1994]. This results in an upper limit on the runtime (and possibly space) complexity of the zoom algorithm: Algorithms that have a quadratic or exponential runtime with respect to the number of nodes in the model are not usable. For example, the runtime is a problem for the graphically distorting techniques (cf. Section 3.1.4) because they are inherently complicated in their implementation and some of them require a significant amount of system time to generate a new image [Leung and Apperley, 1994]. Furthermore, the property that the display of a new layout after a zoom operation should be immediate makes it hard to apply techniques that try to detect overlapping nodes and remove the overlaps because the detection usually implies an exponential runtime because each node has to be checked against each other node.

5.7 Multiple Focal Points

It is often desirable to view multiple areas of a model in detail simultaneously. The zoom algorithm should therefore support multiple focal points [Bartram et al., 1995; Furnas, 1986; Leung and Apperley, 1994; Shoemaker and Gutwin, 2007]. This property is of special importance for relational data where the details of one entity can often not be fully understood without seeing the details of related entities. For example, in the case of models of software systems it is often necessary to examine several subsystems or components and their mutual interconnections concurrently [Storey et al., 1997a]. This also implies that the user must always have full control over all focal points (i.e., he can set or remove a focal point to/from any node in the model all the time). Thus, the system must not close or decrease the size of nodes besides the focal point automatically if the focus changes. Nodes always have to be opened and closed explicitly by the user. Closing or resizing nodes automatically may also result in a bunch of layout operations after a minimal change of the focus by the user. Additionally, hiding nodes automatically if their size falls below a defined threshold my be quite dangerous as it is not obvious for the user whether

the node does not exist at all or is just hidden at the moment. The price that has to be paid for leaving the user in full control is that the visual complexity [Li et al., 2005] of the diagram is not reduced automatically and an increase in the interaction overhead (cf. Section 5.9). This trade-off between how much influence the user should have and how much is done automatically by the tool emerges in almost all computer supported tasks and can usually not be solved easily. For example, in an experimental comparisons between a linear, an overview+detail and a fisheye interface for reading text documents [Hornbæk and Frøkjær, 2003] the subjects commented that they did not like to depend on an algorithm to determine which parts of the document should be readable at the moment

5.8 Smooth Transitions

The transition between the current layout and the new layout that results from a zoom operation can result in the user being suddenly "lost in the diagram". The user usually needs several seconds to re-assimilate to the new layout. Therefore, the transition between the layouts should be smooth to enhance the continuity [Bartram et al., 1995; Noik, 1994]. Animation provides a way to do so by expressing the causality that led to the new layout [Ware, 2004]. An interactive animation shifts some of the user's cognitive load to the perceptual system because the perceptual phenomenon of object constancy enables the user to track structural relationships without thinking about them [Robertson et al., 1991]. The changes in the layout are tracked by the perceptual system so that no re-assimilation is needed afterwards. The technique of interactive animation has been employed by the CAD (computer-aided design) field for years by animating the rotation of complex structures. Animated transitions offer a way to preserve the mental map while adjusting the layout [Bederson and Boltman, 1999; Bridgeman and Tamassia, 2000] and are therefore a lot more than just "eye candy" to increase the visual attractiveness.

Empirical results indicate that animation indeed aids comprehension [Bederson and Boltman, 1999; Gonzalez, 1996; Klein and Bederson, 2005], but there is no evidence that it also aids task performance time [Chui and Dillon, 1997; Donskoy and Kaptelinin, 1997]. The recommended speed for transitions in animated zooming varies between 0.3 and 1.0 seconds, depending on the application domain and the difference between the initial and final layout [Bederson and Boltman, 1999; Card et al., 1991; Klein and Bederson, 2005].

5.9 Small Interaction Overhead

The introduction of a fisheye zoom algorithm into a modeling tool lets the user interactively explore the model and generate different views that are best suited for his current task. The interaction overhead that is required to achieve this effects should be as small as possible and the user should remain in control all the time [Bartram et al., 1995]. The interaction can be

greatly simplified if it is possible to interact directly with the model elements (e.g., by clicking on a node to zoom-in or out) instead of using control elements that are only indirectly connected to the model elements [Scaife and Rogers, 1996]. A user command should then immediately lead to a feedback by the system. Additionally, the user must know all the time where he is in the diagram and where he can go from there (i.e., the user should be aware of all possible navigation paths from the current diagram [Kim et al., 2000]) to effectively interact with the diagram. An example of a problem in the interaction between the user and the model can be found in the graphically distorting techniques (cf. Section 3.1.4): The nonlinear magnification makes the target acquisition cumbersome because objects appear to move away as the focal point approaches them [Gutwin, 2002].

5.10 Minimal Node Size

Some of the focus + context techniques (cf. Section 3.1.4) take the metaphor literally and try to show all the context all the time (i.e., they always show all nodes of the diagram on the screen, though some of them may be visible in an abstracted form). They tighten the property of disjoint nodes (cf. Section 5.2) by stating that "information, whether in a detailed or summary view, should never be occluded" [Bartram et al., 1995], which also makes viewports and scrollbars inappropriate because they do not show the elements that lay outside the currently visible area of the screen. The main problem with these approaches is that individual model elements can become very small if big models (as they often occur for software systems) have to be shown on today's (small) display screens. This makes the interaction cumbersome because of two reasons: First, it can become difficult to hit a small node during the target acquisition (e.g., while clicking on a node to zoom it). And second, the text labels that are used to name nodes and links are not readable anymore if their size becomes too small. Therefore, the size of the nodes must not fall below a defined threshold.

CHAPTER 6

Existing Fisheye Zoom Techniques

Different fields face the same problem of showing large information spaces on small screens, so that a variety of techniques to manage the complexity of diagrams have emerged (see e.g. [Noik, 1994; Spence, 2007, chap. 4] for an overview). This chapter shortly describes the techniques that are most important in the context of graphical models and asseses them against the criteria of Chapter 5. The discussion has a strong bias on the weaknesses of the presented approaches to show why none of them has all the desired properties presented in the last chapter.

All techniques presented in this chapter combine suppression with global distortion (cf. Section 3.3.4) but they originate from different fields: The force-scan algorithm (Section 6.1) has its origins in the field of automatic graph drawing. SHriMP (Section 6.2) was developed to visualize large software structures in the reengineering field. The algorithm of Berner (Section 6.3) has been used for the previous version of the ADORA tool and the Continuous Zoom (Section 6.4) stems from the human-computer interaction field where it has been developed for the user interface of control systems.

Layout Manipulation and Geometric Projection

The presented approaches can be broadly divided in two categories: On the one hand, those that store the model internally in a "fixed canonical coordinate system" [Furnas and Bederson, 1995] (or "normal view" [Noik, 1994]) which is the fully expanded model with all nodes shown in full size and use geometric projections (e.g., scale factors) during the rendering process. The techniques that belong to this category are the Graphical Fisheye Views presented in Section

3.1.4 and the Continuous Zoom. The fixed coordinate system makes it difficult to permit editing operations independent of the current state of the layout. On the other hand are those techniques that store and directly manipulate the current layout, such as the force-scan algorithm, SHriMP and Berner's algorithm. These techniques have only one representation of the model, the one that is currently shown.

6.1 The Force-Scan Algorithm

The force-scan algorithm [Misue et al., 1995] uses the basic concept underlying the "spring algorithms" [Eades, 1984] which are used for the automatic graph drawing (cf. Section 2.3.1) to resolve overlaps between nodes. The main idea is to use a "force" f_{uv} between two nodes u and v that pushes the two away from each other if they overlap. Fig. 6.1 illustrates the force-scan algorithm:

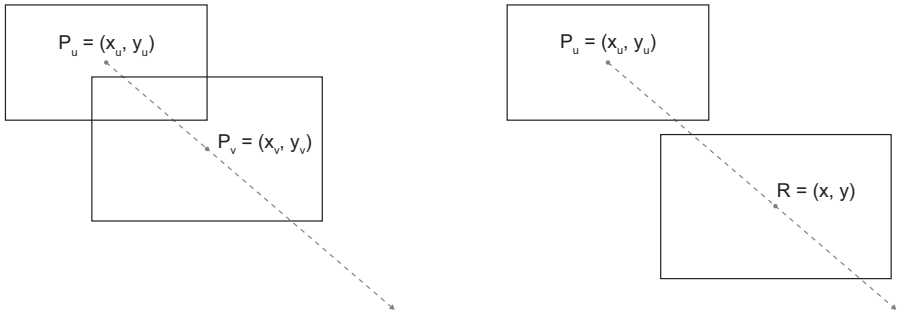


Figure 6.1: The force-scan algorithm

The direction of the force f_{uv} is along the line that connects the centers of the two nodes $P_u = (x_u, y_u)$ and $P_v = (x_v, y_v)$. The magnitude of f_{uv} is set to the difference between the "actual distance" d_{uv} and the "desirable distance" k_{uv} between the two nodes. The actual distance is the Euclidean distance between the two node centers P_u and P_v:

$$d_{uv} = \sqrt{(x_u - x_v)^2 + (y_u - y_v)^2} \tag{6.1}$$

The desirable distance is defined as follows: $R = (x, y)$ is the first point along the line that connects the two node centers P_u and P_v for which either

$$|x - x_u| \geq \frac{w_u + w_v}{2} \qquad \text{or} \qquad |y - y_u| \geq \frac{h_u + h_v}{2} \tag{6.2}$$

holds and where w_u is the width of node u and h_u is the height of node u. The point R is the first point along the line connecting P_u and P_v for which node u with center P_u is disjoint from

node v with center R. The desirable distance k_{uv} is then the Euclidean distance between P_u and R. The magnitude of the force f_{uv} becomes $k_{uv} - d_{uv}$ if the two nodes u and v overlap:

$$f_{uv} = max(0, k_{uv} - d_{uv})l_{uv} \tag{6.3}$$

where l_{uv} is a unit vector in the direction from p_u to p_v. A constant value g can be added to k_{uv} to force a gap of size g between the nodes. Misue et al. call this approach the *push force-scan algorithm* because the force can according to formula 6.3 only be positive which "pushes" the nodes away. Additionally, they propose the *push-pull force-scan algorithm* which allows both positive and negative forces:

$$f_{uv} = (k_{uv} - d_{uv})l_{uv} \tag{6.4}$$

The push-pull force-scan algorithm additionally compacts diagrams which are too sparsely spread out while preserving the general shape of the diagram.

Discussion

The biggest drawback of the force-scan algorithm is that the nodes in its output are not always disjoint. The algorithm has to be repeated until all nodes are disjoint. Therefore, a complex layout can result in a lot of runs of the algorithm. Furthermore, the force-scan algorithm cannot guarantee the stability of the layout because of its heuristic nature: The algorithm calculates all forces and "lets the system go" so that an equilibrium (minimum energy) is reached [Eades, 1984]. However, this equilibrium does not look the same all the time because a different local optimum can be reached. A similar approach is the Orthogonal Dynamic Natural Length Spring (ODNLS) algorithm by Li et al. [2005].

6.2 SHriMP

The SHriMP (**S**imple **H**ierarchical **M**ulti-**P**erspective) visualization technique has been developed by Storey and Müller [1995] to create fisheye views of nested graphs. The technique has its roots in the reengineering field where large existing software systems have to be visualized on small computer screens to explore their structure and browse the source code.

The concept behind SHriMP tries to evenly distribute the distortion throughout the entire graph by uniformly scaling the nodes outside the focal point(s) to avoid too much distortion in some parts of the diagram (as it happens with the Graphical Fisheye Views of Section 3.1.4). Nodes in the graph uniformly give up screen space to allow a node of interest to grow. A node grows by pushing its sibling nodes outward. All nodes are then scaled around the center of the screen

so that they fit inside the available screen space. Therefore, the technique always uses all the available screen space, but does not let the diagram grow beyond this size. Each sibling of the node that grows is pushed outward by adding a translation vector $T = [T_x, T_y]$ to its coordinates:

$$x' = x_p + s(x + T_x - x_p) \tag{6.5}$$

$$y' = y_p + s(y + T_y - y_p) \tag{6.6}$$

where (x', y') are the new coordinates of the sibling, (x_p, y_p) are the coordinates of the center of the screen, s is the scale factor which is equal to the size of the screen divided by the requested size of the screen and (x, y) are the current coordinates of the sibling. The scale factor s can be set to < 1 to shrink a node, that has previously been enlarged. SHriMP employs different translation vectors to preserve either the orthogonality or proximity of nodes (cf. Section 5.3) because it is difficult to preserve both properties using a fixed screen size. Which property is of greater importance can be decided depending on the layout of the diagram and/or the domain.

Preserving Orthogonality

The strategy to maintain the orthogonality relationships first partitions the diagram into nine partitions by extending the boundaries of the scaled node. The translation vector $[T_x, T_y]$ for each sibling node is then calculated according to the partition containing its center. Fig. 6.2 shows the translation vectors for each of the nine partitions.

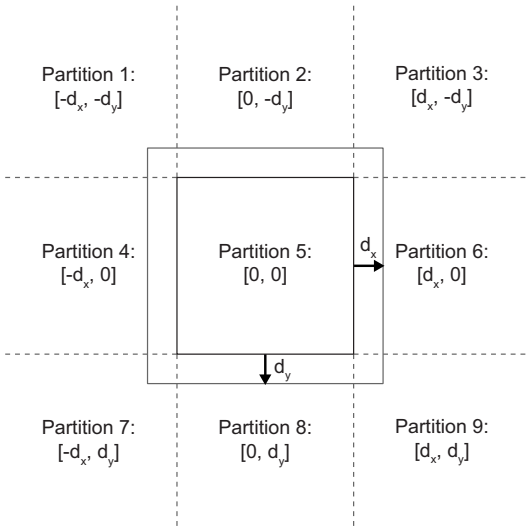


Figure 6.2: Partitions to determine the translation vectors [Storey and Müller, 1995]

For example, a node whose center lies in partition 3 is moved by the translation vector $T = [d_x, -d_y]$, where d_x and d_y are the differences between the new and current width and height of the scaled node. All sibling nodes above and below the scaled node are moved upwards and downwards by the same amount while the nodes on the right and left side are moved right and left, which preserves the vertical and horizontal orthogonality relationships (cf. Section 5.3).

Preserving Proximity

Groups or clusters of nodes that depict certain relationships in the diagram are maintained by preserving the proximity relationships (cf. Section 5.3). SHriMP preserves this property by constraining each sibling node to stay on the line connecting its center and the center of the node being scaled. The resized node pushes the sibling node outwards along this line. Fig. 6.3 shows the calculation of the translation vector for node B if node A grows.

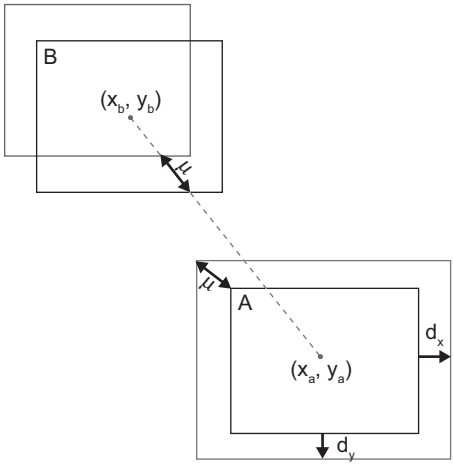


Figure 6.3: Proximity preservation strategy [Storey and Müller, 1995]

The direction of the translation vector for B is equal to the direction of the line that connects the center of B (x_b, y_b) with the center of the scaled node A (x_a, y_a). The magnitude of this vector μ is calculated from the differences between the new width and height of the scaled node A and its previous width and height:

$$\mu = \sqrt{d_x^2 + d_y^2} \tag{6.7}$$

The magnitude μ is the same for all sibling nodes and needs to be calculated only once. The components of the translation vector $T = [T_x, T_y]$ are then calculated by multiplying the magnitude with the direction of the vector:

$$T_x = \mu \frac{x_a - x_b}{\sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}} \tag{6.8}$$

$$T_y = \mu \frac{y_a - y_b}{\sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}} \tag{6.9}$$

Storey and Müller propose also an alternative strategy to preserve the proximity relations which results in a more efficient use of the available screen space. The directions of the translation vectors are calculated the same way, but the magnitude μ is no longer the same for all sibling nodes. Instead, the scaled node moves the sibling nodes according to the displacement of its boundary as it moves along the line connecting its center with the center of the sibling node. Fig. 6.4 illustrates the calculation of the translation vectors in this alternative proximity preservation strategy:

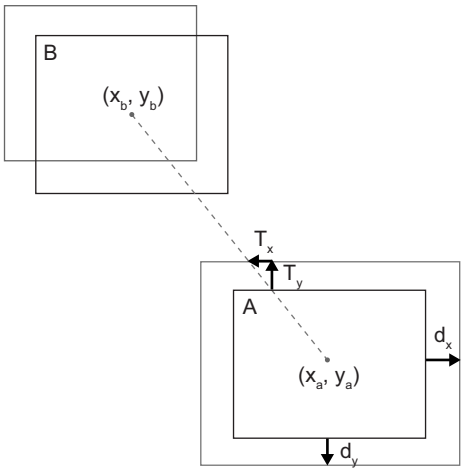


Figure 6.4: Alternative proximity preservation strategy [Storey and Müller, 1995]

The T_x and T_y components of the translation vector are calculated differently depending on the value of the slope m of the line connecting the centers of the enlarged node A and its sibling B ($-d_x$ is used in place of d_x when $x_b < x_a$ and $-d_y$ in place of d_y when $y_b < y_a$):

$$T_x = \begin{cases} \frac{1}{m}(y_b \pm d_y - y_a) + x_a - x_b & \text{if } |m| >= 1, \\ 0 & \text{if } |m| = \infty, \\ \pm d_x & \text{otherwise} \end{cases} \tag{6.10}$$

$$T_y = \begin{cases} m(x_b \pm d_x - x_a) + y_a - y_b & \text{if } 0 < |m| < 1, \\ 0 & \text{if } m = 0, \\ \pm d_y & \text{otherwise} \end{cases} \tag{6.11}$$

Discussion

The main problem with the SHriMP approach is that sibling nodes can overlap if a node that has not previously been enlarged shrinks. Non-overlapping nodes in the original layout can overlap in the adjusted layout if a zoom-out operation is applied to a node before the inverse zoom-in operation. That is usually not a problem for visualizing existing structures (like the software systems SHriMP was originally designed for) because the nodes can always initially be shown in a zoomed-out state so that the user first has to zoom-in before he can zoom-out again. However, it becomes a major problem if the model is built incrementally by the user or is changed while it is (partially) zoomed-out. The internals of a node have to be drawn first before the node can be zoomed-out, so that the the nodes in a modeling tool are always initially zoomed-in, while they can initially be shown zoomed-out in a visualization tool. For example, Fig. 6.5 shows a situation in which the SHriMP approach results in an overlap (the gray shaded area) between the sibling nodes B and C if node A shrinks for (a) the orthogonality preservation strategy, (b) the proximity preservation strategy and (c) the alternative proximity preservation strategy:

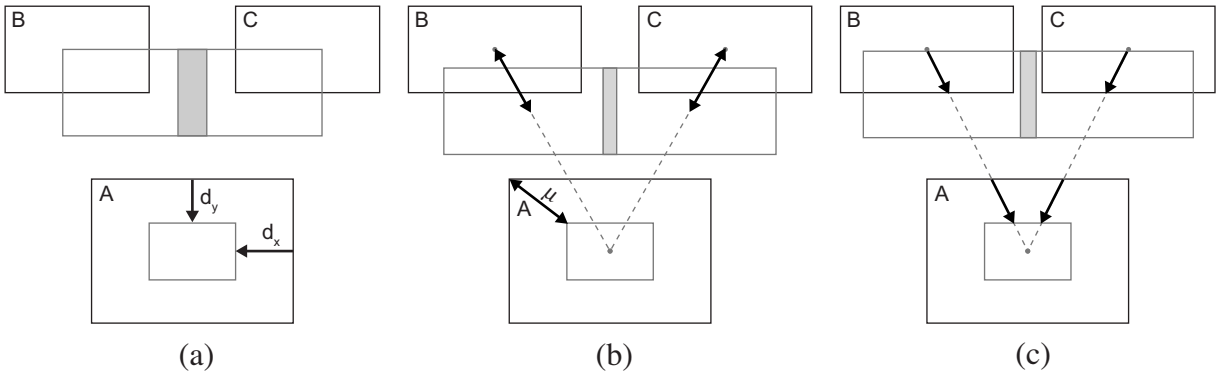


Figure 6.5: Overlapping nodes in SHriMP

SHriMP employs a real fisheye view in which the whole context is visible all the time (potentially with reduced size). Therefore, the technique assumes a fixed maximal screen size which it tries to optimally exploit by scaling the whole diagram. The problem with this approach is that the contextual elements can become very small if the whole diagram is too large (cf. Section 5.10). Additionally, an experimental comparison between program understanding tools [Storey et al., 1997b] has shown that users often do not use the fisheye view feature but instead zoom in to see the details and then zoom out again when more context is needed. Therefore, new implementations of SHriMP combine the Fisheye Zoom with a zoomable user interface (cf. Section 3.1.2).

6.3 Berner

The zoom algorithm proposed by Berner [Berner, 2002; Berner et al., 1998] uses translation vectors to move sibling nodes in a very similar way as SHriMP does. However, it combines two of SHriMP's strategies: the orthogonality and the alternative proximity preservation. Fig. 6.6 illustrates how the translation vectors are calculated in Berner's approach:

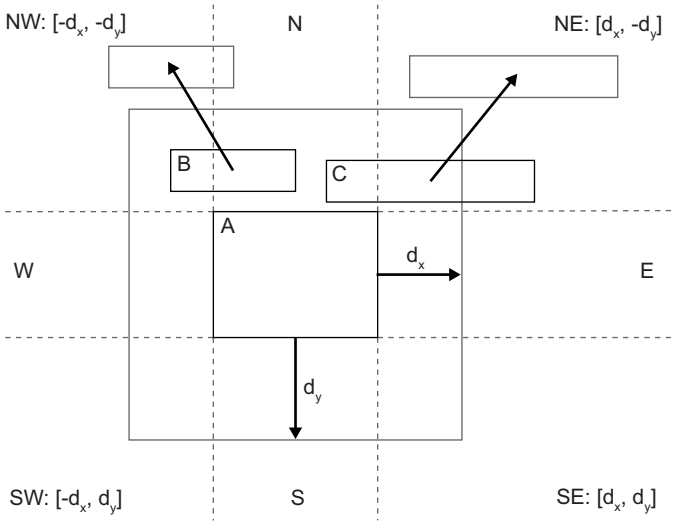


Figure 6.6: Calculation of the translation vectors in Berner's algorithm [Berner, 2002]

The nodes whose center lies in one of the corner partitions (NW, NE, SW and SE) are handled the same way as in SHriMP's orthogonality preservation strategy. Therefore, node C is moved by the two vectors d_x and $-d_y$. Nodes whose center is in one of the other partitions (N, S, E and W) are moved by the translation vectors as they have been calculated by the alternative proximity preservation strategy of SHriMP. The scaled node A moves the sibling node B according to the displacement of its boundary as it moves along the line connecting the centers of A and B.

Discussion

Berner uses a hybrid strategy to better exploit the available space in the diagram. The algorithm shares the limitation of SHriMP that it cannot avoid overlapping nodes if a node is zoomed-out without previously being zoomed-in. However, Berner's approach can also not guarantee that the zoom-in operations produce an overlapping free layout [Marty, 2002]. Fig. 6.7 illustrates the problem: Siblings B and C have to be moved away as A grows to size A'. The translation vector of node C is $[d_x, -d_y]$ because the node's center lies in the corner partition *NE*. The center of node B lies in the top partition N and its translation vector is therefore calculated according to formulas 6.10 and 6.11.

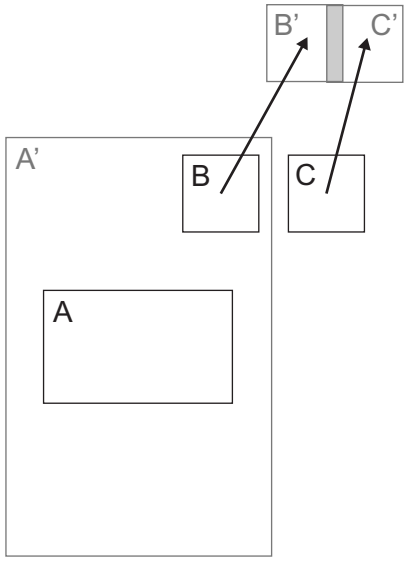


Figure 6.7: Overlapping nodes in Berner's algorithm [Marty, 2002]

Nodes B' and C' overlap in the gray shaded area of the adjusted layout because the differently calculated translation vectors move the nodes closer together. Berner proposes to detect and then automatically resolve this node occlusions. However, the detection can be very costly (cf. Section 5.6) and moving nodes to resolve one occlusion can easily result in new occlusions. Additionally, an automatic occlusion resolution can destroy the secondary notation and the user's mental map.

The basic problem with any technique that uses translation vectors to move the sibling nodes is that it takes only the geometric relations between the zoomed node and its siblings into account but not the relations among these siblings. An independent translation vector is calculated for each sibling node and the node is then moved in isolation without taking surrounding nodes into account. This often results in overlapping nodes.

6.4 The Continuous Zoom

The Continuous Zoom of Bartram et al. [Bartram et al., 1995; Dill et al., 1994] (and its predecessor, the Variable Zoom [Schaffer et al., 1996]) allows the user to control the amount of detail in different areas of the model by opening and closing "clusters" (i.e., nodes that contain other nodes) which toggles the visibility of the contents of the node and allocates more or less space to it. The user can, in addition to this automatic resizing, freely enlarge or reduce the size of any node in the diagram. The Continuous Zoom is presented in greater detail here because our own zoom algorithm presented in Chapter 7 builds on the basic ideas of the Continuous Zoom, namely an interval structure and the concept of scaling interval sizes.

The zoom algorithm combines the initial layout (the so called "normal geometry") with a set of scale factors to produce the "zoomed geometry". The normal geometry is constant while the scale factor for each node is determined by the user. The scale factor of a node whose size remains the same is set to 1.0. The boundaries of the nodes are first independently projected onto the x and y axes, which results in a set of non-overlapping intervals (represented by the dotted lines in Fig. 6.8). A scale factor is then calculated for each horizontal and vertical interval. The scale factor of an interval corresponding to a node projection is the scale factor of this node. The algorithm takes the maximum scale factor of all projected nodes if two or more projections overlap (as A and B on the x axis in Fig. 6.8). The scale factor for gap intervals (i.e., intervals that are not a projection of any node) is set to the maximum scale factor of its neighboring intervals.

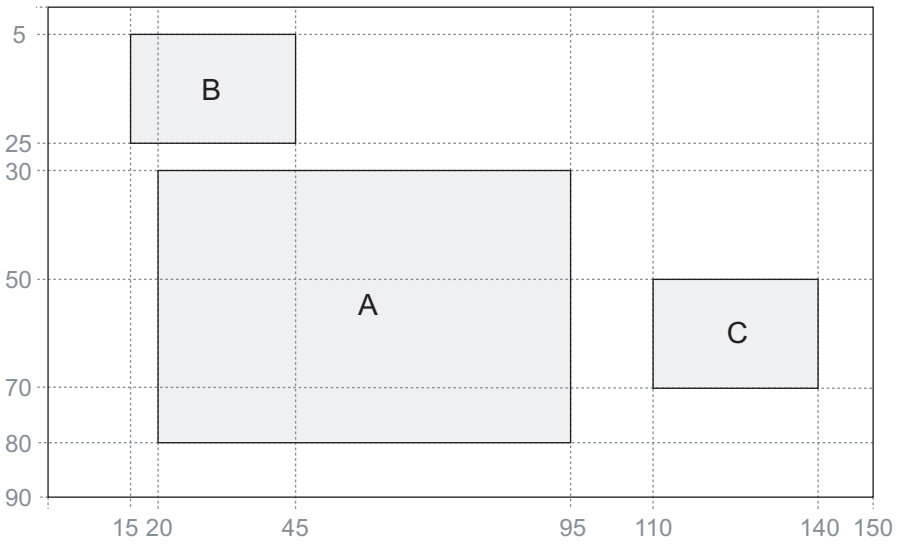


Figure 6.8: Normal geometry of the Continuous Zoom

The algorithm then uses a budgeting process to distribute the available space among nodes. It sums the amount of space that is requested by each node up and then distributes a fixed overall space budget (the display size) according to the size of each request. The total amount of space requested by all nodes inside a cluster in x direction X_{req} is calculated according to 6.12, where x_i is the normal length of the i^{th} interval (the normal length of an interval is the interval's length in the normal geometry) and s_i is the interval's scale factor.

$$X_{req} = \sum_i x_i s_i \tag{6.12}$$

As an example, lets assume that the size of node A in Fig. 6.8 is scaled down by the factor 0.5. The accumulated normal lengths of the horizontal and vertical intervals in Fig. 6.8 are shown by the numbers on the boundaries of the intervals. The scale factor for nodes B and C is 1.0, because their size remains the same. Thus, the total space that is requested in the x direction is:

$$X_{req} = 15 * 1 + 5 * 1 + 25 * 1 + 50 * 0.5 + 15 * 1 + 30 * 1 + 10 * 1 = 125 \tag{6.13}$$

The third horizontal interval (the one with the normal length of 25) is scaled by the factor 1.0 because nodes A and B are both projected to this interval and the zoom algorithm takes the bigger of the two scale factors which is in the example the one of B. The size of a node must be adjusted if the size of at least one of its children changes. Therefore, the scale factor of a parent node is "inherited" from its children's scale factors:

$$S_p = \frac{X_{req}}{X_p} \tag{6.14}$$

Where S_p is the scale factor and X_p the normal width of the parent node. The space request of each node is propagated upwards until the root node is reached. The size of the root node is fixed (limited by the screen size). The parent node of A, B and C in Fig. 6.8 is the root node so that its scale factor in x direction becomes:

$$S_{root} = \frac{125}{150} = 0.83 \tag{6.15}$$

The normal width of the root node in Fig. 6.8 in x direction of 150 is the sum of all the horizontal intervals' normal lengths. The scale S_i of each interval in the hierarchy is divided by the scale factor of the root node S_{root}, so that the size of the whole diagram remains constant. The zoomed length x_i^* of an interval is then the normal length x_i of the interval multiplied by this modified scale factor:

$$x_i^* = x_i \frac{S_i}{S_{root}} \tag{6.16}$$

For the first horizontal interval in Fig. 6.8 (for which the normal length is 15 and the scale factor 1.0) this becomes:

$$x_1^* = 15 \frac{1.0}{0.83} = 18 \tag{6.17}$$

The length of the horizontal and vertical intervals containing a node constitute the total space available to that node (the so called zoom hole). The final size of a node has to be calculated separately because the size of the node may differ from its zoom hole (because of differences in the scale factors):

$$L_k^* = L_k \frac{S_k}{S_{root}} \tag{6.18}$$

Where L_k^* is the zoomed width, L_k the normal width and S_k the scale factor of node k. This results in the new width of node A:

$$L_A^* = 75\frac{0.5}{0.83} = 45.18 \tag{6.19}$$

The nodes are repositioned according to the location of their center points inside the corresponding interval. The aspect ratio of the nodes may change because the zoomed width and height are calculated separately. The smaller of the two scale factors can be applied in both directions to held the aspect ratio constant (which may in turn result in an inefficient use of space). Applying the just described algorithm accordingly to the horizontal and vertical intervals of Fig. 6.8 results in the final situation shown in Fig. 6.9. Note that the decrease of A's size results in an increase of the size of B and C. The zoom holes of the nodes are depicted by the gray shaded areas in the figure.

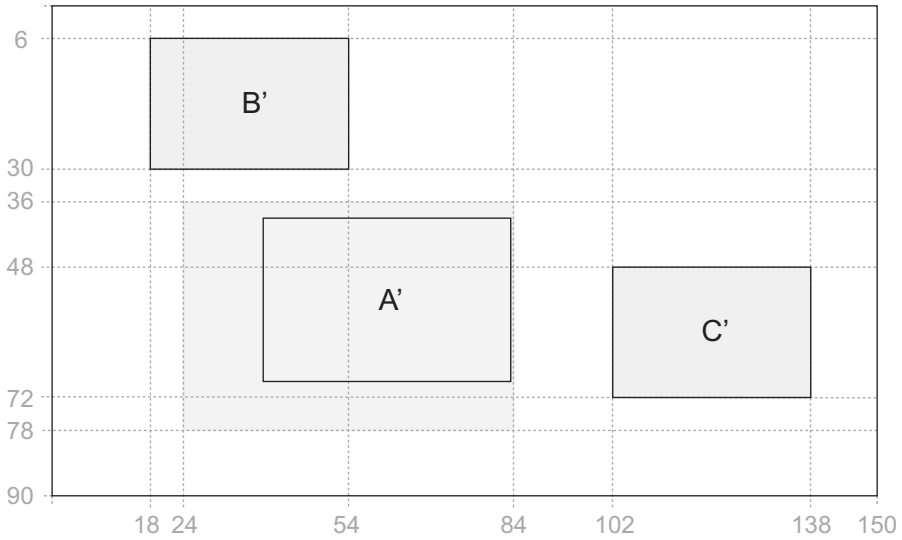


Figure 6.9: Node A has been scaled down by factor 0.5

Automatic Node Sizing

The Continuous Zoom relies on the user in deciding on the size of a node. However, early user testing indicated a desire for more optimal sizing in which the node opens to its maximum size without any extra manipulation. Thus, the focus of the approach has shifted over time towards reducing the amount of size manipulations for the user with the ultimate goal of automating the node sizing. As a possible solution, Bartram et al. [1995] suggest to use the Degree of Interest (cf. Section 3.3.3) for the automatic node sizing. The DOIs are dynamically calculated based on a node's a priori importance, its current state (e.g., normal operation or alarm state in the case of

a control system) and its proximity to interesting nodes (i.e., nodes with a high DOI). As a result, neighbors of interesting nodes get more space than nodes that are further away, which results in the fisheye effect.

Discussion

The basic assumption of the Continuous Zoom that the size of the diagram (which is determined by the available display space) remains fixed, results in a global behavior of the zoom algorithm: changing the scale factor of just one nodes changes the size of all nodes (as can easily be seen by comparing Fig. 6.8 and Fig. 6.9). This global nature can have a negative influence on the user's mental map as shown by some early user reactions [Dill et al., 1994]: "Some users found the resulting motion in peripheral parts of the display to be distracting when they were trying to focus or setting the size of a particular node. Furthermore, after setting the size of one node, it was annoying to have it change as a result of resizing another node somewhere else." Such interdependencies between node sizes is inevitable because the diagram has a fixed size.

An additional problem that results from the assumption that the screen has a fixed size and is always completely visible is the fact that most graphical models are just too big (or the screens too small) so that they cannot be completely shown at once on a screen even when an fisheye technique is applied (cf. Section 5.10). This problem emerges with all "In-Situ Magnification" techniques [Berner, 2002] which assume a fixed diagram size such as SHriMP (cf. Section 6.2) or the distortion-oriented techniques (cf. Section 3.1.4).

And finally, the basic geometric projection of the Continuous Zoom that combines the fixed normal geometry and the scale factors to generate a geometric projection, the zoomed geometry, makes it difficult to permit editing operations independent of the current state of the scaled geometry. While the editing takes place on the zoomed geometry, the changes have to be applied back through the geometric transformations onto the normal geometry.

CHAPTER 7

Zoom Algorithm

The purpose of a zoom algorithm is to produce a new layout if the size of a node changes because its inner structure is hidden or shown. It takes an existing layout (i.e., the position and size of a set of non-overlapping nodes) as well as a new size for an existing node η as input and calculates a new layout. The size of η is updated to the new size and the position of the siblings is adjusted to the new size of η. The zoomed-in or zoomed-out node η (or one of its siblings) pushes the boundaries of its parent outward or pulls them inward, respectively. The parent in turn pushes or pulls its siblings, which is done recursively until the root node is reached. Thus, the zoom algorithm is applied recursively on all the direct or indirect parents of η.

The presented algorithm works on an interval structure and employs the concept of scaling interval sizes[1] and is therefore similar to the Continuous Zoom by Bartram et al. [1995] (cf. Section 6.4). However, our approach does not assume a fixed basic layout, does not globally scale the whole model and uses different scaling functions. The improved zoom algorithm has all properties described in Chapter 5, thus overcoming the limitations of the other approaches presented in Chapter 6. We assume a layout with rectangular representations of the nodes. However, this is no limitation as the boundaries of nodes with arbitrary shape can be represented by a rectangle. Parts of the algorithm have already been published in [Reinhard et al., 2007, 2008].

[1] This idea goes back to the very early focus+context techniques like the Bifocal Display [Spence and Apperley, 1982] and the Rubber Sheet [Sarkar et al., 1993].

7.1 Data Structure

The algorithm works on an *interval structure* which is constructed by projecting the node boundaries on the X- and Y-axes. The spaces between the grid lines created by the projections are called *intervals*. Each node in the hierarchy which can be further decomposed has an interval structure of its own (with the childrens' projections to the axes as intervals). Fig. 7.1 shows the projection of the three nodes A, B and C on the axes of the coordinate system. Y_1 to Y_7 are the *vertical intervals*, whereas X_1 to X_7 are the *horizontal intervals*. By $len(\delta)$ we denote the *length* of a specific interval δ. The length of a horizontal interval is the difference between its right and left boundary, while the length of a vertical interval is the difference between its bottom and top boundary.

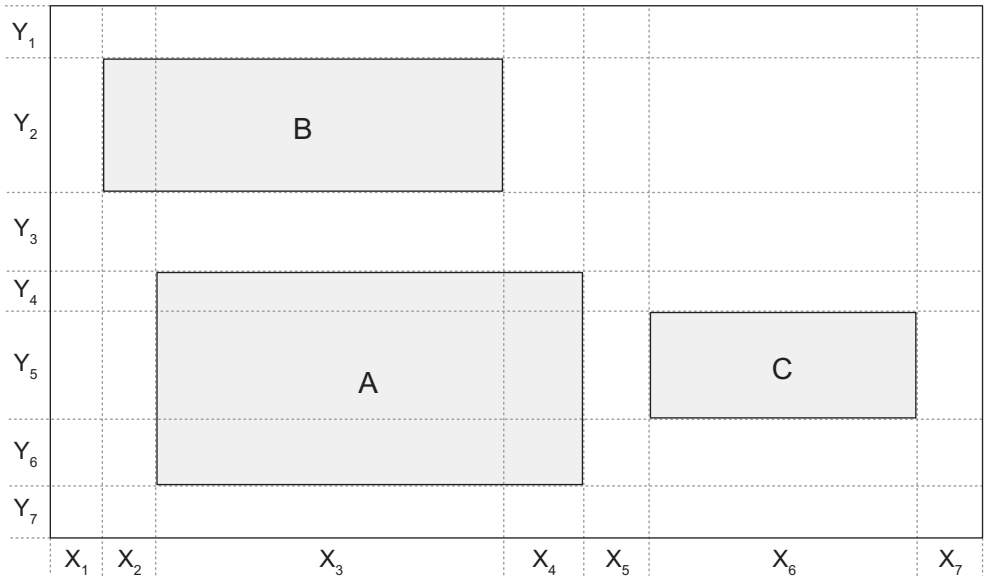


Figure 7.1: Interval structure

We distinguish between *node intervals* that are projections of nodes (or part of nodes) and *gap intervals* which are projections of the gaps between nodes. For example, Y_2, Y_4, Y_5 and Y_6 are vertical node intervals in Fig. 7.1, while Y_1, Y_3 and Y_7 are the vertical gap intervals. We denote the set of horizontal node intervals of a specific node η by $proj_x(\eta)$ and the set of vertical node intervals of a specific node η by $proj_y(\eta)$. In Fig. 7.1 we have: $proj_y(A) = \{Y_4, Y_5, Y_6\}$ and $proj_x(A) = \{X_3, X_4\}$. The horizontal and vertical node intervals of a node η form together the node intervals of η: $proj(\eta) = proj_x(\eta) \cup proj_y(\eta)$.

Intervals cannot overlap by definition, even though multiple nodes can be projected to the same node interval. For example, nodes A and B in Fig. 7.1 have both interval X_3 as horizontal node interval (i.e., $X_3 \in proj_x(A) \wedge X_3 \in proj_x(B)$). Each node is represented by a *zoom hole*[2] in the

[2]We follow here the terminology of Bartram et al. [1995].

interval structure. The zoom holes of nodes A, B and C are shown by the gray shaded rectangles in Fig. 7.1. The width of a zoom hole is equal to the accumulated length of the node's horizontal node intervals, while its height is equal to the accumulated length of the node's vertical node intervals.

The order of the vertical and horizontal intervals is always preserved if a zoom operation is applied to the interval structure even though the lengths of the intervals may change. This property is important to guarantee an overlapping-free layout after a zoom operation (cf. Section 5.2).

Construction of the Interval Structure

The interval structure of a node is incrementally build by adding nodes when they are inserted into the structure's parent node. The horizontal and vertical intervals are completely independent so that they are also updated separately when a node is inserted. The algorithm to adjust the horizontal intervals in the structure once a new node is inserted works as follows (the vertical intervals are adjusted accordingly):

1. The interval δ_{left} that contains the left boundary of the node is identified. This can be achieved by a sequential or binary search since the intervals are stored in a sorted sequential list.
2. Interval δ_{left} is split at the position of the node's left boundary (if this boundary lies within the interval). The interval that has been newly created by this split is added to the list containing all horizontal intervals of the structure and δ_{left} is set to this new interval.
3. The interval δ_{right} that contains the right boundary of the node is identified.
4. A reference to the zoom hole of the inserted node is added to all intervals between the left δ_{left} (inclusive) and right δ_{right} (exclusive) interval. Additionally, a reference to each of these intervals is added to the horizontal intervals of the zoom hole of the node.
5. The right interval δ_{right} is split at the position of the node's right boundary (if this boundary lies within the interval). The interval that has been newly created by this split is added to the list containing the horizontal intervals of the structure.
6. A reference to the zoom hole of the inserted node is added to the right interval δ_{right} and a reference to the right interval δ_{right} is added to the zoom hole of the inserted node.

Removing a Node from the Interval Structure

The horizontal and vertical intervals of an interval structure have to be adjusted if a child is removed from a node so that the interval structure still reflects the internal structure of this parent node. The horizontal and vertical intervals are managed independently so that they can also be

adjusted independently. The algorithm for the horizontal intervals is described in the following (the one for the vertical intervals works accordingly).

1. The algorithm has to remove the zoom hole from the interval structure because the representation of a node in the structure is the node's zoom hole. The algorithm iterates over all horizontal intervals that are covered by the zoom hole $z(n)$ of node n to be removed and executes the following steps for each interval δ in the iteration:

2. The left δ_{left} and right δ_{right} neighbor of the current interval δ are identified. This is achieved by simply following the references to the previous and next interval that are stored in each interval.

3. The reference to the zoom hole $z(n)$ is removed from the current interval δ.

4. The following steps are executed if the current interval δ has a left neighbor δ_{left} (an interval does not have a left neighbor if it is the first horizontal interval of the structure): The algorithm checks whether the current interval δ and its left neighbor δ_{left} have both no projected zoom holes or the two sets containing the zoom holes that are projected δ and δ_{left} are identical. The current interval δ is merged with its left neighbor δ_{left} if one of these two conditions holds by the following steps:

   (a) The right boundary of δ_{left} is set to the right boundary of the current interval δ.

   (b) The reference of δ_{left} that points to δ_{left}'s right neighbor is set to δ_{right}.

   (c) The current interval δ is removed from the ordered set containing all the horizontal intervals of the interval structure.

   (d) The reference from any zoom hole to the current interval δ is removed.

   (e) The reference of δ_{right} to δ_{right}'s left neighbor is set to δ_{left}.

   (f) δ_{left} becomes the new current interval.

5. Step 4 is then repeated accordingly for the right neighbor δ_{right}.

7.2 Zoom Operations

Our zoom algorithm scales the horizontal and vertical intervals of the interval structure independently if the size of one of the node changes because it is zoomed-in or out. It is, according to the classification of Leung and Apperley [1994], a Cartesian technique (as opposite to the Polar techniques of Section 3.1.4). Carpendale et al. [1997a] classify these techniques as "step orthogonal" distortions. The Cartesian approach has two major problems: First, it creates only small distortions but at the cost of leaving a lot of unused space. And second, the resulting layout can contain unintended ("ghost") foci [Leung and Apperley, 1994] and clusters that are not related

to the data itself and could lead to misinterpretations [Carpendale et al., 1997a]. The following sections explain how our zoom algorithm addresses these problems.

We start in Section 7.2.1 with the simpler of the two operations, the zooming-in of a node which increases the number of details. The sibling nodes have to be moved away to provide the space that is needed by the expanding node. The inverse operation, a zooming-out that collapses the node and decreases the level of detail is presented in Section 7.2.2.

7.2.1 Zooming-In

The zoom algorithm has to increase the length of some of the intervals if node η is zoomed-in because η needs more space. The intervals are adjusted by first calculating two independent scale factors which are then applied to the vertical and horizontal node intervals of η. Adjusting the lengths of the node intervals of η repositions the siblings of η. The new size of η can be a constant value or it can be calculated with respect to the space that is needed by η's children. The two independent scale factors $s_x(\eta, \eta')$ and $s_y(\eta, \eta')$ are calculated as follows:

$$s_x(\eta, \eta') = \frac{width(\eta')}{width(\eta)} \tag{7.1}$$

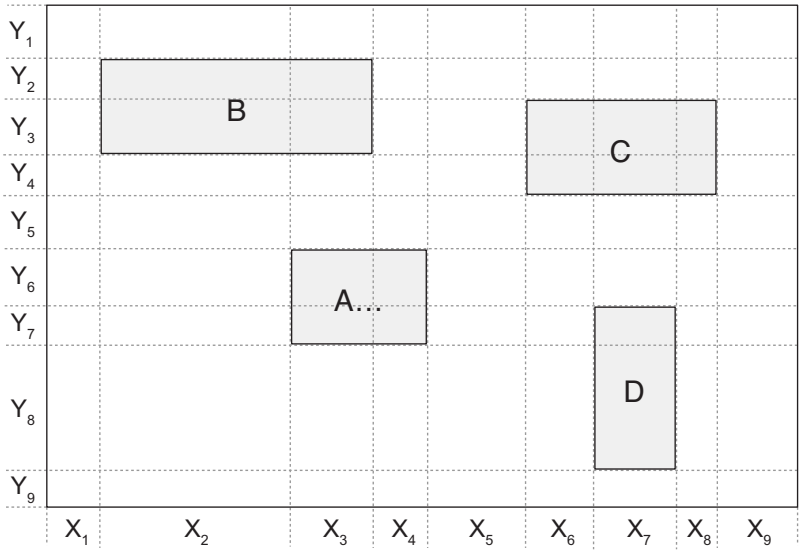
$$s_y(\eta, \eta') = \frac{height(\eta')}{height(\eta)} \tag{7.2}$$

$width(\eta)$ and $height(\eta)$ denote the width and height of the boundaries of node η. The node before zooming-in is represented by η, whereas η' is the node after zooming-in. The factors s_x and s_y are greater than one because the node is enlarged during a zoom-in. These zoom factors are then applied to the node intervals of η, so that the following predicates are satisfied

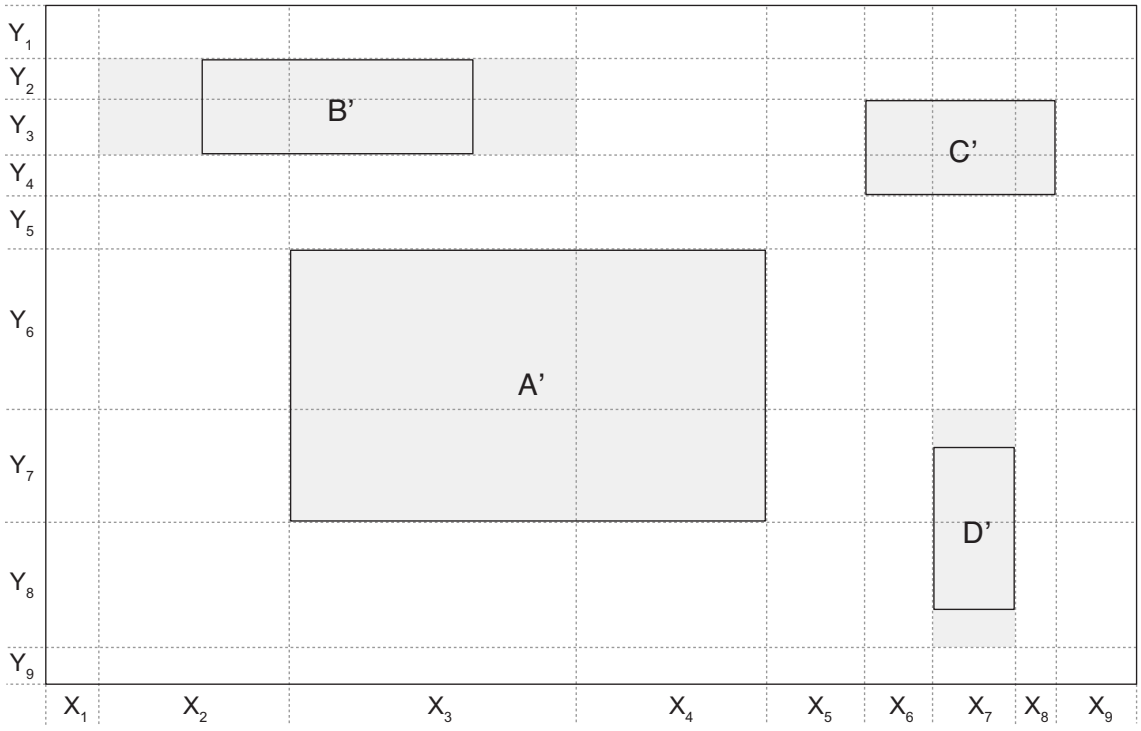
$$\forall \delta \in proj_x(\eta) : len(\delta') = len(\delta) * s_x(\eta, \eta') \tag{7.3}$$

$$\forall \delta \in proj_y(\eta) : len(\delta') = len(\delta) * s_y(\eta, \eta') \tag{7.4}$$

where δ' represents the interval δ after the zooming. For example, Fig. 7.2a) shows the initial situation with a zoomed-out node A. In a subsequent step, node A is zoomed-in, which is illustrated in Fig. 7.2b). For zooming node A, the scale factors s_x and s_y are calculated by taking the boundaries of node A in Fig. 7.2a) as initial size and the boundaries of node A' in Fig. 7.2b) as new size. The length of the node intervals of A is adjusted by these scale factors (the vertical intervals Y_6 and Y_7 by s_y and the horizontal intervals X_3 and X_4 by s_x). The length of the remaining intervals (i.e., those that are not in $proj(A)$) is not changed.



(a)



(b)

Figure 7.2: Zooming-in node A

Fig. 7.2b) shows the situation after node A has been zoomed-in. The length of the vertical intervals Y_6 and Y_7 and the horizontal intervals X_3 and X_4 increased while the lengths (but not necessarily the boundaries) of the other vertical and horizontal intervals remained constant. The two nodes A and D are both projected to the vertical interval Y_7 (i.e., $Y_7 \in proj_y(A) \wedge Y_7 \in proj_y(D)$). The length of this interval is scaled by the factor s_y because the height of node A increases. In Fig. 7.2b), the zoom hole $z(D)$ of node D (i.e., the area defined by the vertical projection of node D' or $proj_y(D') = \{Y_7', Y_8'\}$) is no longer congruent with node D'. Moreover, the width of the zoom hole of node B' in Fig. 7.2b) is bigger than the width of node B' because the length of the vertical interval X_3 increased to X_3' due to the zooming-in of node A.

As shown for nodes B' and D' in Fig. 7.2b), a zoom hole's size can be bigger than the size of the node. The size of the zoom hole and the node may also be equal in size, as for nodes A' and C'. The zoom hole $z(C')$ in Fig. 7.2b) is congruent with node C' because none of the node intervals of C is also a node interval of A (i.e., $proj_x(C) \cap proj_x(A) = \emptyset \wedge proj_y(C) \cap proj_y(A) = \emptyset$). The concept of zoom holes is used to avoid unintended foci in the adjusted layout. A sibling node becomes an additional focus that was not intended by the user if its size increases because one of its node intervals, which is also a node interval of the zoomed node, is scaled. Thus, the size of the sibling nodes must not change if a node is zoomed to avoid these unintended foci. This can be achieved by decoupling the actual size of a node from the space that it currently occupies in the interval structure.

Position of a Node Inside the Zoom Hole

Our zoom algorithm centers a node inside its zoom hole if the boundaries of the zoom hole are bigger than the boundaries of the node (as for example nodes B' and D' in Fig. 7.2b)). An alternative would be to maintain the relative position of the node's center inside the interval [Bartram et al., 1995] in order to preserve this property of the mental map. Enlarging the node to the size of the zoom hole is not an option because it results in unintended foci.

7.2.2 Zooming-Out

Zooming-out is simple when the zoom-out operation immediately follows a zoom-in operation of the same node. In this case, zooming-out can be accomplished by just reversing the zoom-in algorithm: Zooming-out node A' in Fig. 7.2b) results in the layout shown in Fig. 7.2a) because the scale factors which are applied to the node intervals of A' are the inverse of those that were applied to the node intervals of A during the zooming-in. However, in the general case, zooming-out is not that easy. The algorithm must not generate overlapping nodes (cf. Section 5.2) and the layout should be stable under multiple zooming-in and zooming-out operations (cf. Section 5.4). Overlapping nodes cannot occur when using an interval structure, as long as the zoom-out algorithm preserves the order of the intervals. However, the stability property does not hold if the zooming-out operation is the first zoom operation (i.e., it is not following a zooming-in

operation). The zoom algorithm has to be extended to guarantee the stability of the layout in such situations. For example, Fig. 7.3a) shows an initial layout, in which node A is zoomed-out in the subsequent step. The zoom operation results in the new node A' as shown in Fig. 7.3b). The scale factors s_x (for the horizontal intervals X_2, X_3 and X_4) and s_y (for the vertical intervals Y_4, Y_5 and Y_6) are now less than one because the size of node A decreases. The vertical interval Y_5 and the horizontal interval X_3 cannot be scaled down by the scale factors s_y and s_x because the length of the intervals would be smaller than the height of node C and the width of node B respectively. The zoom algorithm therefore has to maintain a *minimal length* for each interval that should be scaled down.

The minimal length l_{min} of an interval δ can be calculated as follows. Each zoom hole z_i defines for each interval it is projected to a property *real length* l_{real}. The value of l_{min} for interval δ is the maximal real length of all zoom holes z_i that are projected to δ:

$$\begin{aligned} &\exists l \in \mathbb{N} : \forall k \in \mathbb{N} : 0 < l \leq m \wedge 0 < k \leq m \\ &\Rightarrow l_{real}(z_l, \delta) \geq l_{real}(z_k, \delta) \wedge l_{min}(\delta) = l_{real}(z_l, \delta) \end{aligned} \tag{7.5}$$

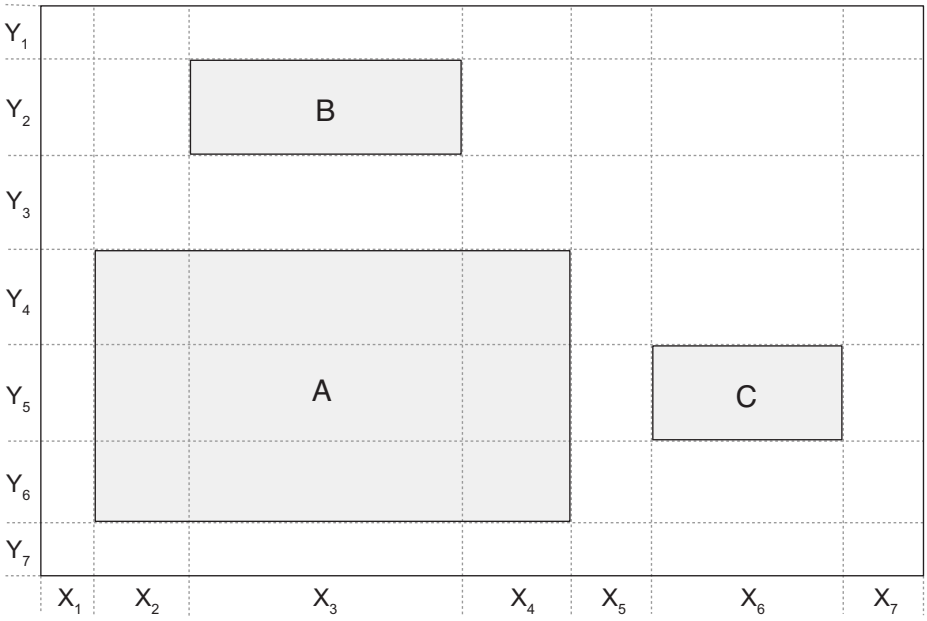
where m is the number of zoom holes that are projected to δ. The real length for each interval is initially set to the length of the interval (i.e., $l_{real}(z, \delta) = len(\delta)$). It is set to the current real length of the interval scaled by s_x or s_y if node η is zoomed-in or zoomed-out ($z(\eta)$ denotes the zoom hole of node η and δ' the interval after the zoom operation):

$$\forall \delta \in proj_x(\eta) : l_{real}(z(\eta), \delta') = l_{real}(z(\eta), \delta) * s_x \tag{7.6}$$

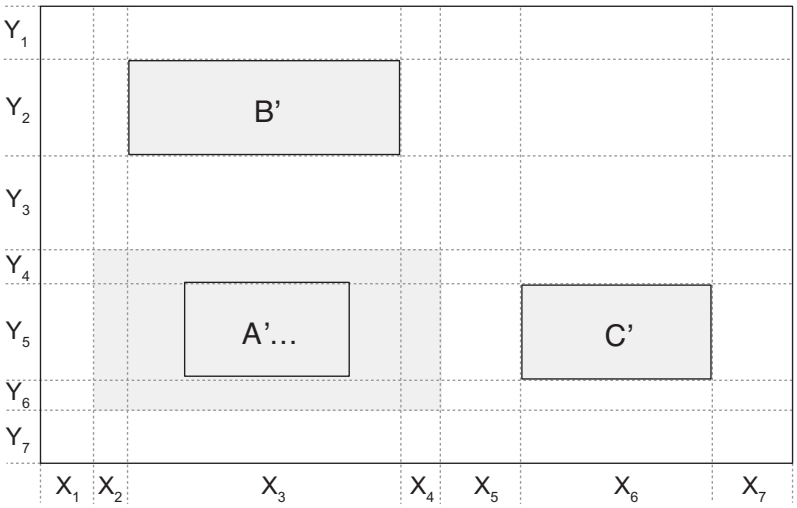
$$\forall \delta \in proj_y(\eta) : l_{real}(z(\eta), \delta') = l_{real}(z(\eta), \delta) * s_y \tag{7.7}$$

For example, the real length of the zoom hole of node A' for the vertical interval Y_5' (i.e., $l_{real}(z(A'), Y_5')$) in Fig. 7.3 is obtained by multiplying the scale factor s_y with the length of interval Y_5. The real length of the zoom hole of node C' for Y_5' (i.e., $l_{real}(z(C'), Y_5')$) is the height of node C' (which is the initially set value that has not been changed during the zoom-out operation). The minimal length of interval Y_5' is the real length of the zoom hole of node C' because this length is bigger than that of the zoom hole of node A'.

The zoom algorithm has to store the real length for each interval in the zoom hole because the scaling up that may be done by a following zoom-in operation has to be done on this real length. The resulting node would be bigger than the original node A if node A' in Fig. 7.3b) is scaled up by multiplying the length of the intervals Y_5' and X_3' with the scale factors obtained from A' and A because Y_5 and X_3 are not scaled down during the zoom-out operation. This is due to the fact that our zoom algorithm always works on the existing interval structure and does not construct a new structure or adjust the structure to the new size of the nodes.



(a)



(b)

Figure 7.3: Zooming-out node A

Fig. 7.3 illustrates the problem of additional superfluous white space that can result from a Cartesian zooming technique: There is a lot more white space around the zoomed-out node A in Fig. 7.3b) than in Fig. 7.3a) because the intervals X_3 and Y_5 cannot be scaled down as they are additionally projections of nodes B and C respectively. One way to mitigate this problem is to scale the neighboring gap intervals together with the node intervals. Thus, the intervals Y_3 and Y_7

as well as X_1 and X_5 are scaled too when node A shrinks so that less space around the zoomed-out node is wasted. However, such a dense layout can also be a problem (cf. Section 5.1) so that an additional scaling of gap intervals is not always desirable. But it can for some specific tasks such as the filtering of nodes (cf. Chapter 8) increase the quality of the layout considerably.

Zoomed-Out Node Size

The new size that a node should have after a zoom-in operation has to be set to the space that is required to show the children of the node. This size is equal to the size of the node's internal interval structure. The calculation of the new size after the inverse operation, a zoom-out, does not have such an obvious solution. The abstracted node hides all its children so that its shape becomes a simple rectangle whose size can be set to any value. The size calculation has do be done only once when the node is zoomed-out for the first time and should be done automatically to release the user from this task [Bartram et al., 1995]. The zoomed-out node size can then be stored for subsequent zoom-out operations. A stored zoomed-out node size makes it also possible that the user can adjust this size which is then preserved.

The calculation of the zoomed-out node size must adhere to two constraints: First, the size of the node should be reduced if its internals are hidden, so that the overall size of the diagram is reduced (cf. Section 5.1). And second, the new size of the node must not fall below a minimal threshold (cf. Section 5.10). Different approaches may be suitable for different applications so that it makes sense to leave the size calculation to the application code. Some possible techniques are the following:

- All nodes are set to the same **constant size** if they are zoomed-out. The main problem with this technique is that the overall shape of the node gets lost. This can be problematic because the aspect ratio between the width and height of the node may be important for the mental map (cf. Section 5.3).

- The new size of a zoomed-out node can be set to the **size of the node label** to make sure that the label is not truncated. However, the problem that the overall shape of the node is lost remains.

- The overall shape of the node can be preserved if the **original node is scaled**. The critical point with this technique is the value of the scale factor. Using the same constant scale factor for all nodes can result in situations where the size of big nodes is not reduced sufficiently while the new size of small nodes falls below the defined threshold. Therefore, a different scale factor has to be calculated for each zoomed-out node.

  A scale factor that sufficiently reduces the size of nodes while taking care of their minimal size can be calculated as follows: The minimal node size is defined by a constant minimal width $width_{min}$ and a constant minimal height $height_{min}$ for all nodes. The larger of the two scale factors which are calculated from the current width $width$ and the minimal width

$width_{min}$ and the current height $height$ and the minimal height $height_{min}$ respectively is then used to scale the node:

$$s = max(\frac{width_{min}}{width}, \frac{height_{min}}{height}) \tag{7.8}$$

- Setting the new size of the zoomed node to the **size of its zoom hole** seems to be the obvious solution because the zoom hole defines the minimal size the node can be reduced to in the current layout. The size of the zoom hole can be seen as the "natural" size for the node in the current layout, which also reduces the number of additional bendpoints that have to be added to the links. However, the node has first to be scaled down to the "target size" which then defines the size of the zoom hole as the zoom hole results from the scaling of the node's intervals with a specified scale factor. Additionally, this may result in situations in which the size of the node is not changed at all if the node is zoomed-out, but changes later when another node is zoomed-out.

- A completely different approach to calculate the new size of a zoomed-out node is to **encode semantic information about the node in its size**. For example, the size of the node can give some hints about its internal complexity or its degree of interest (cf. Section 3.3.3).

7.3 Bottom-up Zooming

The zoom algorithm has to adjust the layout recursively in a bottom-up way because the nested boxes that represent the composition form a tree structure. The basic concept is illustrated by means of Fig. 7.4:

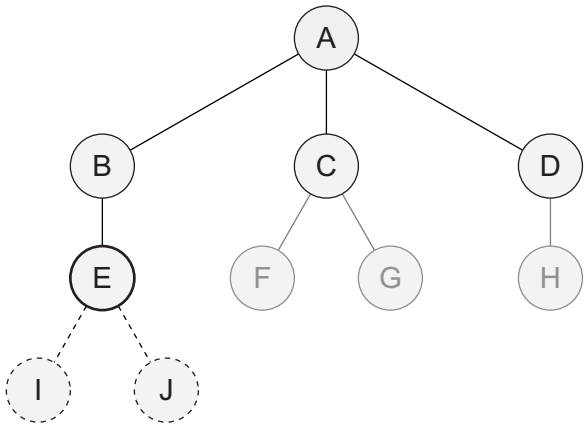


Figure 7.4: Bottom-up zooming in the hierarchy

Lets assume that node E in Fig. 7.4 is zoomed-out. Nodes I and J are removed from the current diagram as they are children of the now abstracted node E. The size of the zoomed-out node E shrinks which also reduces the size of E's parent B. Node E has no sibling nodes so that no nodes inside B have to be moved. The zoom algorithm now has to adjust the positions of B's siblings C and D as the size of B decreases. The reduced size of B and the new position of C and D in turn reduce the size of node A which is in this case the root node and therefore represents the whole diagram. Thus, the zoom algorithm is applied recursively on all direct and indirect parents of the zoomed node as well as on the siblings of these ancestors. Note that nodes F, G and H can be ignored by the zoom algorithm because they are neither an ancestor of the zoomed node E nor a sibling of one of the ancestors. Thus, the zoom algorithm can only partially exploit the partitioning of the decomposition structure because of the nested box notation: Changing the layout on a lower level of the hierarchy entails additional changes on higher levels.

CHAPTER 8

Dynamically Generated Views

The zoom algorithm, as described so far, exploits the hierarchical structure of a model to generate different views by showing only abstract representations of some model elements. However, the underlying concepts of the zoom algorithm can also be used in a more general way in order to adjust the layout if an individual node or a set of nodes is hidden in the current diagram. This filtering mechanism employs a secondary means to reduce the size and complexity of a diagram. Different views that display different facets of a system (structure, data, behavior, user interaction etc.) can be generated from a single integrated model by showing model elements of certain types only [Seybold et al., 2003]. Hence, a tool can be used to generate different views automatically instead of forcing the modeler to create views himself by drawing a multitude of diagrams of different types, as required in UML for example (cf. Sections 2.2.2 and 3.2). The combination of a fisheye zoom with such a view generation mechanism in a tool makes the use of comprehensive integrated modeling languages feasible.

The fisheye zoom which employs a vertical abstraction by suppressing all direct and indirect children of a node and the view generation mechanism which hides individual nodes or nodes of a certain type to abstract horizontally share the same underlying implementation. However, they can be used independently: Individual nodes or sets of nodes can be filtered regardless of the current zoom state of the diagram and zoom operations can be applied on any dynamically created view. Additionally, the generation of different views by filtering nodes of a specific type can be used on flat (i.e., non-hierarchical) models too.

8.1 Node Filters

The specific kind of the views that are supported and which nodes should be shown in these views depends on the application the zoom algorithm is used in. However, the underlying idea remains the same: Generate views that show different parts of a model for a specific purpose (i.e., pragmatic criterion of a model [Ludewig, 2003]). Some exemplary views are the following:

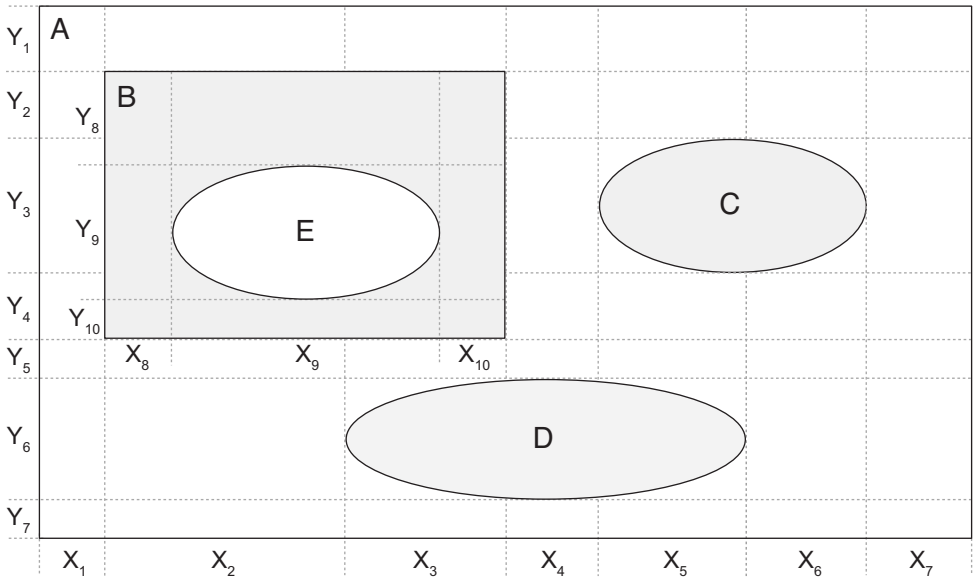
- Views can be generated by showing or hiding all model elements that are instances of a specific language construct. Different modeling facets (e.g., structure, data, behavior and user interaction of a system) can then be shown by generating different views from a single underlying model [Seybold et al., 2003].
- All model elements that collaborate to fulfill a specific task can be shown in one view so that the user can easily see those elements without being overwhelmed by a multitude of elements that do not have anything to do with this task. Examples of such tasks are a specific interaction between a set of objects or the behavior that is described by a (sub)set of states in a statechart [Harel, 1987].
- It is often desirable to see all nodes that are somehow related to the node that is currently investigated (cf. Section 5.7). The filtering mechanism can be used to show all nodes that are somehow (e.g., by a link) related to a selected node.

8.2 Filter Operation

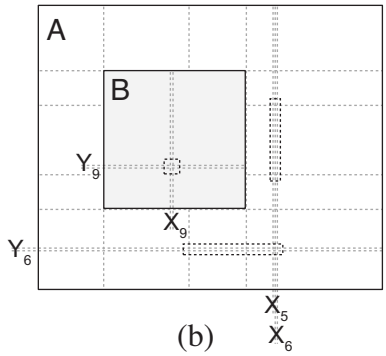
The zoom algorithm described in the last chapter can without modification be used for the filtering of nodes. The positions of the remaining nodes have to be adjusted if a single node or a set of nodes is hidden. The zoom algorithm does exactly that, it adjusts the layout (i.e., the position and size of the remaining nodes) if the size of a node changes. Thus, we can reuse the algorithm by reducing the size of the node which should be hidden to a minimal size which in turn compacts the whole diagram. The size of the node is reduced to the smallest possible size which is $width(\eta') = 1$ and $height(\eta') = 1$ (it has to be > 0 in Equations 7.1 and 7.2 of Section 7.2.1 so that the scale factors s_x and s_y are > 0). The zoom algorithm is then used to recursively adjust the layout to this new (minimal) size of the node.

The example of Fig. 8.1a) shows two nested interval structures: one for node A and one for A's child B. Nodes C, D and E are of a different type than nodes A and B (as indicated by the different shapes) and should therefore be hidden in a subsequent view. The horizontal intervals of A's interval structure are X_1 to X_7 and its vertical intervals Y_1 to Y_7. The interval structure of node B consists of the horizontal intervals X_8 to X_{10} and the vertical intervals Y_8 to Y_{10}. We first consider the filtering of node E: The size of E is reduced to $width(E') = 1$ and $height(E') = 1$. The horizontal and vertical node intervals of E, $proj_x(E) = \{X_9\}$ and

$proj_y(E) = \{Y_9\}$ respectively, are scaled down to E's new size. The length of the remaining horizontal and vertical intervals (X_8, X_{10} and Y_8, Y_{10} respectively) remains the same. Reducing the lengths of the intervals X_9 and Y_9 reduces the size of B's interval structure which in turn reduces the size of node B.



(a)



(b)

Figure 8.1: Filtering nodes C, D and E

Fig. 8.1b) shows the situation after nodes C, D and E have been hidden. The lengths of intervals X_9 and Y_9 have been scaled to 1 because the size of the node that is projected to these intervals has been set to $(1, 1)$ and no other node is projected to these two intervals. Node E is too small to be visible anymore but its position and size are indicated by the dashed rectangle within node B in Fig. 8.1b). Nodes C and D are hidden by using the same mechanism. However, the vertical interval Y_3 cannot be scaled down completely while the height of node C shrinks because it is also a projection of node B (i.e., $Y_3 \in proj_y(C) \wedge Y_3 \in proj_y(B)$) whose height is also reduced but not by the same amount. The horizontal node intervals of C, X_5 and X_6, can both be fully scaled

down so that the zoom hole of node C becomes the area marked with the second dashed rectangle in Fig. 8.1b). The same holds for the horizontal node intervals of D: The length of the horizontal interval X_3 cannot be fully scaled down because node B is also projected to this interval. Fully scaling the horizontal interval X_4 removes the gap between nodes B and C as the length of the interval becomes 1. Completely removing a gap between two nodes should be avoided as the white space between nodes is an important part of the secondary notation (cf. Section 2.5). Therefore, we introduce a *minimal distance* between two nodes and the constraint that the length of an interval that keeps two nodes apart must not fall below this threshold. The last horizontal node interval X_5 of node D can be reduced to 1 as it is only a projection of C and D which are both scaled down to $(1, 1)$. While the minimal distance takes care that the zoom algorithm does not remove too much white space from the diagram, it may often be desirable to remove more white space than just the one that disappears with a hidden node. This can be achieved by additionally scaling the gap intervals next to the node which is hidden. For example, the length of interval X_7 in Fig. 8.1 can be reduced while node C is removed from the diagram in order to reduce the white space to the right of node B in Fig. 8.1b).

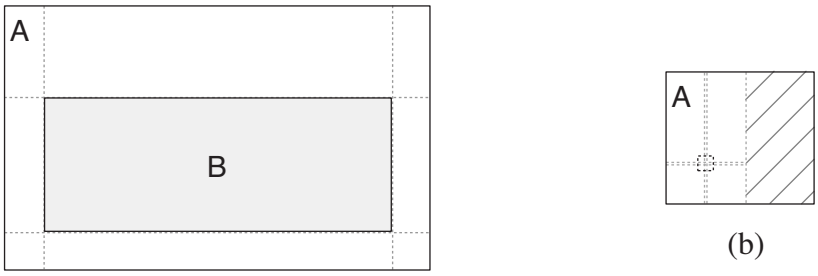
Hidden nodes can be shown again by setting the size of the node to its initial size and letting the zoom algorithm adjust the layout. Setting the size of nodes C, D and E in Fig. 8.1b) to their initial size and then using the zoom algorithm to adjust the position and size of the remaining nodes results again in the initial layout of Fig. 8.1a). The size a node had before it was hidden has to be stored so that the node's size can be restored once the node is shown again. Fortunately, this size (i.e., the size the node has if it is shown) is already stored in the node's interval structure. Additionally, using the size of the interval structure makes it possible to set the size of the node to the correct value regardless of whether some of the children of the node are currently zoomed-out or hidden. Hiding node E in Fig. 8.1a) results in the new size of its parent B that is shown in Fig. 8.1b). If node B is then hidden and shown again in subsequent steps, the resulting layout should be that of Fig. 8.1b) which is achieved by setting B's size to the size of its interval structure. This works also correctly if the filtering of node E is undone before B is shown again: There is no immediate effect if E is shown again, because its parent B is currently hidden, but showing B in a subsequent step results again in the initial situation shown in Fig. 8.1a).

Those fisheye techniques that are based on translation vectors like SHriMP or Berner's approach (cf. Sections 6.2 and 6.3 respectively) cannot be used for such a node filtering mechanism because the filtering of a node is an extreme case of a zoom-out operation that is applied before the inverse zoom-in operation: The new size is the minimal size the node can have and the node has to be hidden (which is equivalent to the zoom-out) before it can be shown again (the equivalence of the zoom-in). The vector-based techniques cannot guarantee that the nodes in the adjusted layout are still disjoint after a zoom or filtering operation in such situations.

Minimal Node size

Setting the size of the parent node to the size of its interval structure if one (or multiple) of its children are hidden can result in situations where the property of the minimal node size (cf. Sec-

tion 5.10) does not hold anymore. For example, node B in Fig. 8.2a) should be hidden. The size of B is reduced to the smallest possible size (i.e., $width(B') = 1$, $height(B') = 1$) and the zoom algorithm is used to adjust the layout to B's new size. The fact that the width of node B is only marginal smaller than the width of the parent node A results in the situation that the width of A's interval structure becomes smaller than the width of the predefined minimal node size. As a consequence, the filtering mechanism has to make sure that the width or height of a node whose children are hidden does never fall below the threshold defined by the minimal size of the node. Fig. 8.2b) shows the situation where the width of the interval structure of A is smaller than the width of node A. The only problem with this approach is that the user can insert a new node (or move an existing node) into the area inside node A that is no longer covered by the interval structure (the hatched area in Fig. 8.2b)). Thus, the zoom algorithm has to make sure that the interval structure is expanded accordingly if a node is inserted into or moved to this area, because the user must always be able to edit the model regardless of the current zoom or filter state (cf. Section 5.5).



(a) (b)

Figure 8.2: Minimal size of node A if node B is hidden

Zooming-out versus Filtering

A zooming-out operation suppresses all children of a selected node while the filtering mechanism hides an individual node or a set of nodes. Thus, zooming-out can be seen as a special instance of the filtering concept as it hides a set of nodes too (namely all children of a selected node). Using the filtering mechanism to hide all children of a node that is zoomed-out makes the calculation of the zoomed-out node size (cf. Section 7.2.2) obsolete as the size of the node can just be set to the new size of the node's interval structure (i.e., the size defined by the shape that remains if all children of the node are filtered). This simplifies the whole concept because there is only one operation (i.e., filtering) required and the size of the zoomed-out node is calculated implicitly. However, this approach has two problems: First, the gap intervals between the children are not scaled while the nodes are hidden and may therefore avoid a substantial decrease of the node's size (which is desired to get a compact layout). The zoom algorithm has to scale the gap intervals to avoid that the white space between the children bloats the size of a zoomed-out node. The

second problem is that the size of the zoomed-in and zoom-out node are tighter coupled so that they cannot be changed independently anymore. Changing the size of the zoomed-out node does also change its zoomed-in size as the change is done on the interval structure.

CHAPTER 9

Editing Support

The nested box notation used to depict hierarchical relations lends itself to some kind of sophisticated tool support (like the fisheye zooming and view generation mechanism described in the previous chapters) but the need for tool support, especially for the editing, is also significantly larger than for ordinary (flat) graphs. A change in one part of the diagram is propagated all the way up in the hierarchy until the root node is reached. The user has to manually adjust the size of all direct and indirect parent nodes and possibly the location of their siblings to provide the required space if a new node is inserted. Other editing operations such as removing or moving a node also result in a lot of tedious manual work for the user. Thus, a layout technique which automatically expands or contracts parent nodes if a node is inserted or removed is highly valuable [Seybold et al., 2003] as it reduces the interaction overhead (cf. Section 5.9) significantly. Extending the fisheye and view generation technique to support the user while editing the diagram moves it from a visualization technique to a more general "layout adjustment strategy" [Misue et al., 1995; Storey and Müller, 1995]. The extended zoom algorithm should not only permit editing operations (cf. Section 5.5) but actively support them. A major advantage of the zoom algorithm, presented in Chapter 7, is that it can easily be used for automatic layout adaption if the diagram is edited by for example inserting or deleting a node.

User Control and Automation

While deciding on the amount of the layouting task that should be done automatically or manually by the user, one inevitably faces a trade-off between the two properties of a small interaction overhead (cf. Section 5.9) and the freedom to use the secondary notation (cf. Section 2.5). The

user has to spend a lot of time on graphical or layouting tasks instead of on the modeling of a system if no or only rudimentary tool support is provided for the editing. The other extreme, a full automatic layouting technique, as it is for example employed by the automatic graph drawing algorithms (cf. Section 2.3.1), can neither support the secondary notation of a diagram nor the preservation of the mental map in case of a layout change. Therefore, a layout adjustment strategy that has the properties of Chapter 5 has to be situated somewhere between a "dumb" drawing program that lets the user draw shapes at will and a fully automatic approach that treats the user as a passive bystander. It should assist the user by unburdening him from the tedious layouting tasks while he stays in charge all the time.

9.1 Inserting Nodes

The editing operation with the most obvious need for tool support is the insertion of a node. The requirement that a new node must not overlap with existing nodes (cf. Section 5.2) often results in the need to adjust the position of a large part of the existing nodes. Fig. 9.1 shows the situation, where node D is inserted into a diagram that already contains nodes A, B and C.

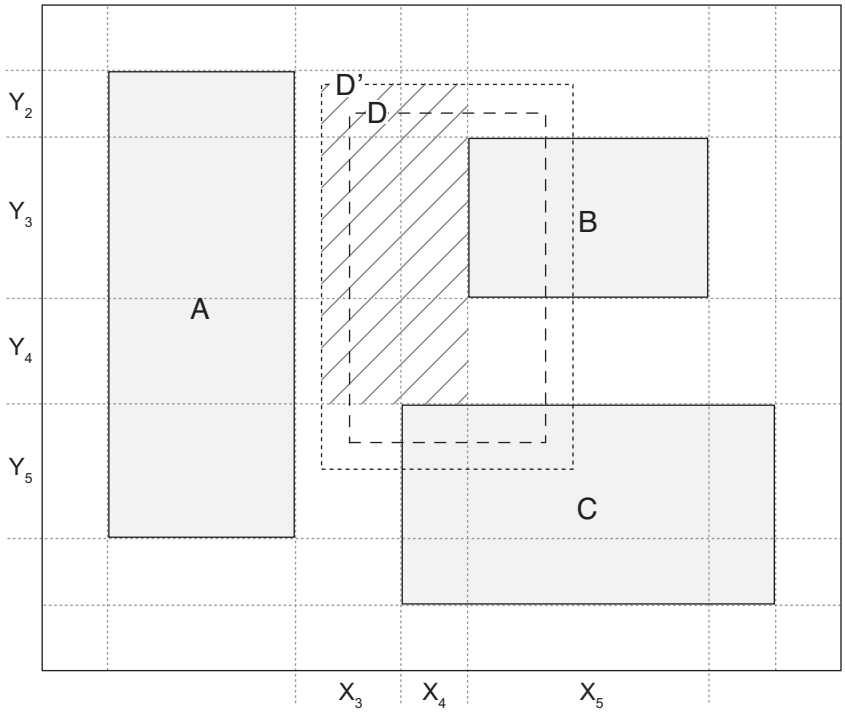


Figure 9.1: Inserting a new node

The boundaries the node should be inserted with are expanded by a predefined value to assure that there is a minimal gap between the new and existing nodes in the final layout. The dotted rect-

angle marked with D' in Fig. 9.1 shows this *expanded node* while the dashed rectangle marked with D represents the original boundaries of node D. The interval structure is then searched for the space that the new node can maximally occupy without moving any sibling node away. The resulting *free space* is depicted by the hatched area in Fig. 9.1. The node is then inserted with the size of this free space by using the algorithm described in Section 7.1. The zoom algorithm is in the next step used to increase the size of the node to its expanded size. The resulting situation for the insertion of node D can be seen in Fig. 9.2.

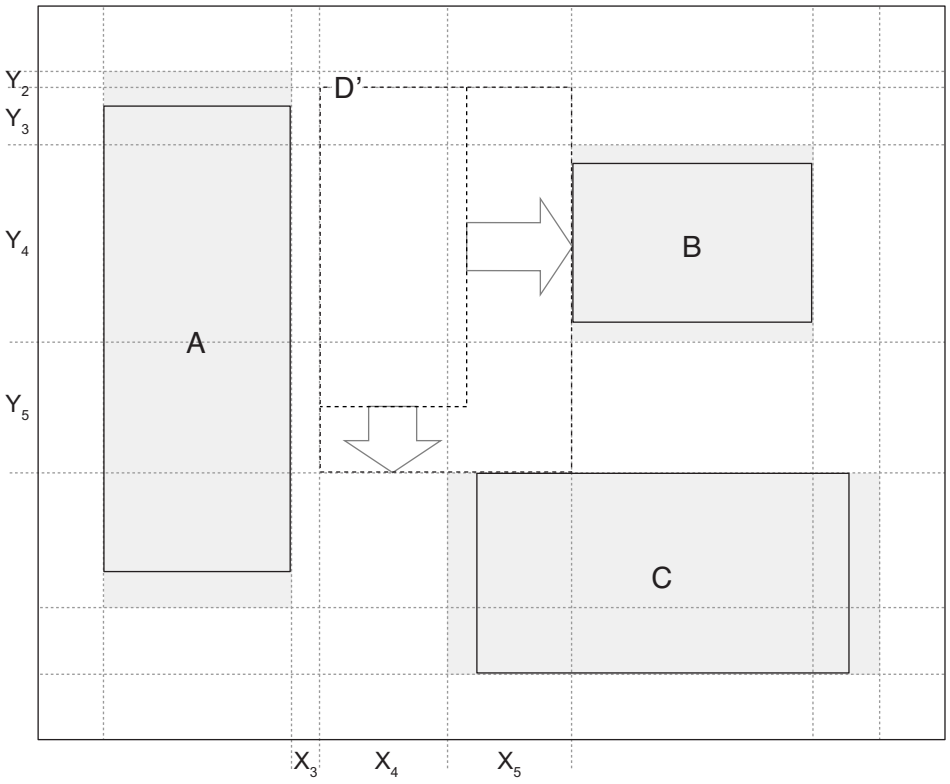


Figure 9.2: Situation after the expanded node D' has been inserted

The space that is required to insert the expanded node D' has been created by the zoom algorithm by moving the existing nodes B and C away (B towards the left and C towards the bottom). The node has been inserted with the size shown by the smaller dotted rectangle and then expanded to the size of the bigger dotted rectangle. The vertical interval Y_2 of Fig. 9.1 has been split into the two intervals Y_2 and Y_3 in Fig. 9.2 (see Section 7.1 for a detailed explanation of how exactly the interval structure is updated if a new node is inserted). The zoom algorithm has then scaled the length of the vertical and horizontal node intervals of D', namely Y_3, Y_4, Y_5 and X_4, X_5 respectively to the length shown in Fig. 9.2. Note that the scaling of the intervals has also changed the size of the zoom holes of nodes A, B and C (shown by the gray shaded areas) so that their boundaries are no longer congruent with the boundaries of the nodes they represent. Finally, the expanded node is removed from the interval structure (without scaling the intervals) and then

inserted again with its original size. This results in a minimal gap between the new and existing nodes as the original size is smaller than the expanded size. The final situation for the insertion of node D into the diagram of Fig. 9.1 is shown in Fig. 9.3. The just presented technique can also be used to provide additional white space in the diagram without actually inserting a node.

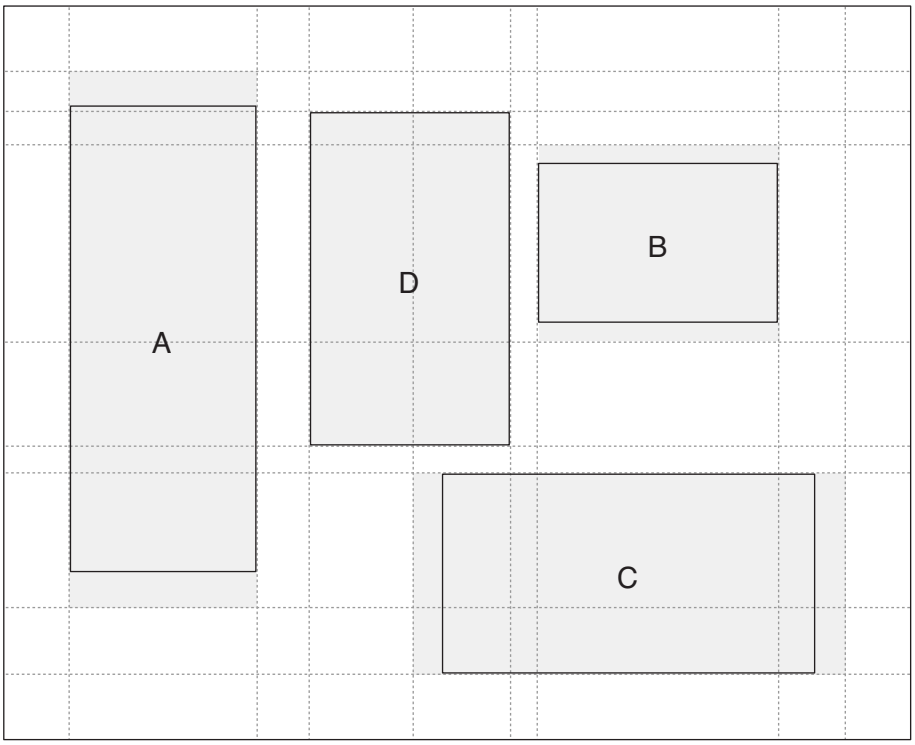


Figure 9.3: Final situation after node D has been inserted

9.1.1 Calculation of the Free Space

The maximal size a new node can be inserted with is calculated before the zoom algorithm scales the intervals to provide the space that is required by the new node. The impact on the current layout (i.e., the amount of change) is minimized if the node is inserted with the size it can maximally have in the current layout because the difference between this free space and the original size becomes minimal. Additionally, the origin (i.e., the top left corner) of the calculated free space should be as close as possible to the origin of the boundaries the new node should be inserted with. The calculation of the free space can be realized by using the interval structure and a sequence of simple geometric and set operations. This algorithm to calculate the minimal space is explained by means of Fig. 9.4.

The basic idea is to first iterate over all horizontal intervals the new node lies in to find existing

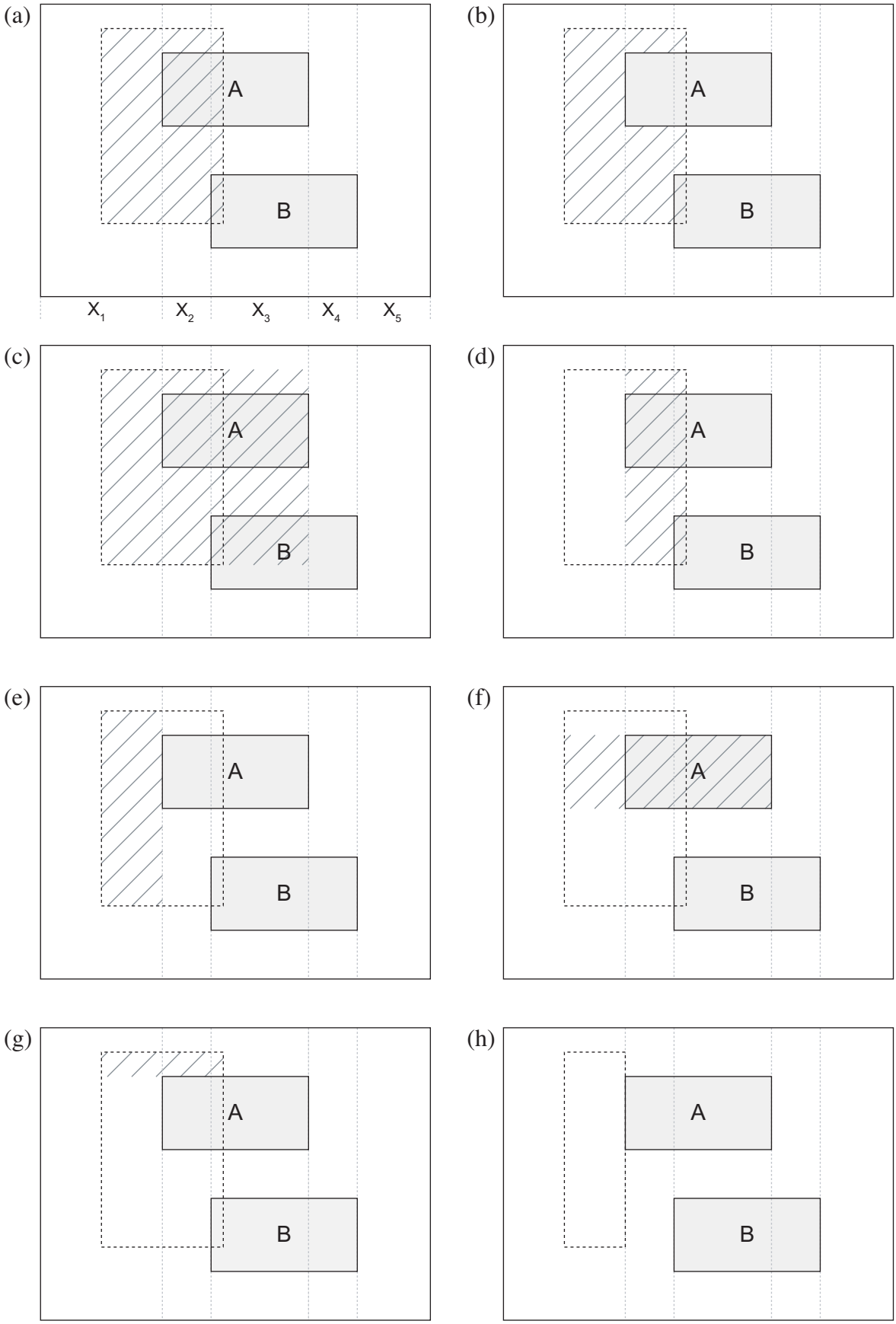


Figure 9.4: Calculation of the maximal available free space

nodes that overlap with the new node. Second, the rectangles whose origin lies as close as possible to the origin of the new node and whose area covers as much of the area of the new node as possible without an overlap with an existing node are calculated. Finally, the biggest of these rectangles is selected. The detailed steps are the following:

1. The *free space* is set to the rectangle which represents the space that is required. Fig. 9.4a) shows the situation where the maximal free space that is available for a new node in the diagram already containing nodes A and B should be calculated. Initially, the free space (represented by the hatched area) is set to the boundaries of the new node (the dotted rectangle).

2. The algorithm then iterates over all horizontal intervals covered by the boundaries of the new node to check for overlaps with existing nodes. For the situation of Fig. 9.4a) these are the intervals X_1, X_2 and X_3. All nodes (or more precisely all zoom holes of the nodes) projected to the current interval in the iteration are checked for an overlap with the free space. The first hit in the example of Fig. 9.4a) is node A in interval X_2. The following steps are executed for each node for which such an overlap exists.

3. The overlapping existing node is subtracted from the free space. Node A is subtracted from the free space in Fig. 9.4b), which results in the hatched polygon. The subtraction is a simple set operation: $D = FS \setminus A$, where D is the resulting set, FS the set representing the free space and A the set representing node A. The free space is then set to the enclosing boundaries of the calculated area. If the resulting polygon is a rectangle, the solution has been found. The following steps are executed only if the resulting area D is not rectangular, as it is the case for Fig. 9.4b).

4. The enclosing rectangle of the union between the free space and the bounds of the overlapping node is calculated (the hatched area in Fig. 9.4c)). This area is then horizontally reduced to the part that covers the free space as well as the existing node. The resulting area is shown by the hatched rectangle in Fig. 9.4d).

5. The intersection between the resulting rectangle and the free space is now subtracted from the polygon of step 3, which results in the hatched rectangle in Fig. 9.4e). The resulting rectangle is the first of the two candidate solutions.

6. Step 4 is now repeated in vertical direction: The enclosing rectangle of the union between the free space and the bounds of the overlapping node is calculated and then vertically reduced to the part that covers the free space as well as the existing node, as shown in Fig. 9.4f).

7. The intersection between this resulting rectangle and the free space is now subtracted from the polygon of step 3, which results in the hatched rectangle in Fig. 9.4g). The resulting rectangle is the second of the two possible solutions.

8. The calculated free space is now set to the rectangle with the bigger area. For the example, this is the first solution shown in Fig. 9.4e). The new node can then be inserted with this size as shown in Fig. 9.4h).

The next iteration of step 2 checks interval X_3 for an overlap between the free space and nodes A and B as they are both projected to this interval. However, no overlap is detected because the free space has been reduced to the dotted rectangle of Fig. 9.4h) during the first iteration.

9.1.2 Inserting Nodes Into a Partially Hidden Model

The view generation mechanism described in Chapter 8 places some extra challenges on the algorithm that adjusts the layout in case of an editing operation because some of the existing nodes that have to be moved may be hidden at the moment the operation takes place. For example, the newly inserted node may overlap with an existing yet currently invisible node. However, our zoom algorithm handles this situation without any modifications because the hidden node is still in the interval structure (with a minimal size as described in Section 8.2) and is moved away like any other (visible) node.

9.2 Removing Nodes

The zoom algorithm should remove the empty space that becomes available if a node is removed from the diagram to fulfill the requirement of a compact layout (cf. Section 5.1). The layout adaption works the other way round as for the insertion of a new node: The size of the node that should be removed is first reduced to a minimal size to compact the layout and the node is then removed from the interval structure. The zoom algorithm faces the same task of compacting the layout as if a node is hidden to generate a specific view (cf. Section 8.2). The only difference between the two operations is that the node remains in the interval structure if it is hidden while it is removed from the interval structure if the node is deleted.

The removal of a node is illustrated by means of Fig. 9.5 where node B is removed from the diagram. The size of the node is reduced to the smallest possible size of $width(B) = 1$ and $height(B) = 1$. The zoom algorithm is then used to scale the affected intervals to adjust the layout. The intervals Y_3 and Y_4 are the vertical node intervals of B (i.e., $proj_y(B) = \{Y_3, Y_4\}$) while X_4 and X_5 are its horizontal node intervals (i.e., $proj_x(B) = \{X_4, X_5\}$). The length of these intervals is scaled when the size of node B is reduced as indicated by the two arrows. The length of intervals Y_3 and X_5 cannot be reduced because they are also projections of nodes A and C respectively which remain in the diagram (see Section 7.2.2 for a detailed discussion). Finally, the intervals that represent node B are removed from the interval structure (or more precisely, merged with their neighbored intervals as described in Section 7.1).

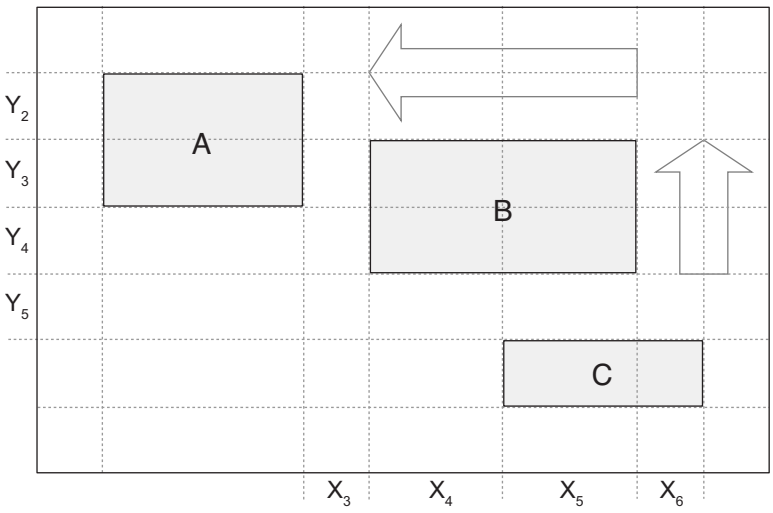


Figure 9.5: Removing node B from the diagram

Fig. 9.6 shows the final situation after node B has been removed. The length of intervals Y_4 and X_4 have been reduced to 1 to remove the resulting empty space from the diagram. This moves nodes A and B closer together and thereby reduces the overall size of the diagram. The vertical intervals Y_2 and Y_3 as well as Y_4 and Y_5 have been merged because these intervals are no longer needed after node B has been removed. The same holds for the merging of the horizontal intervals X_3 and X_4 as well as X_5 and X_6.

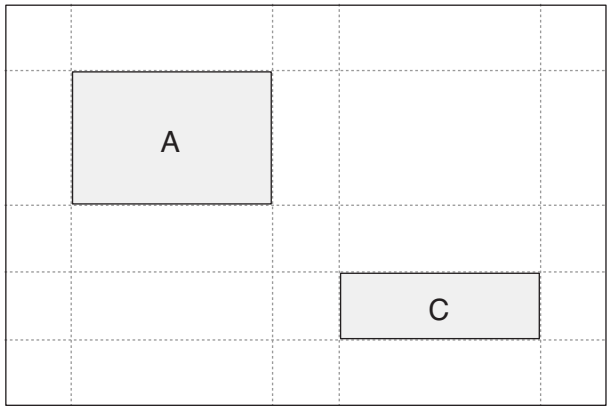


Figure 9.6: Final situation after node B has been removed

The layout can be further compacted during the removal of a node by not only scaling the node intervals but also their neighboring gap intervals (cf. Section 7.2.2). For example, consider again the situation of Fig. 9.5 where not only the vertical node intervals Y_3 and Y_4 but also the vertical gap interval Y_5 is scaled when node B is removed. The scale factor for gap intervals has to be

calculated separately because the one that is calculated for the node intervals reduces their length to the minimal length. The scale factor can for example be set to 0.5 so that the length of the gap intervals is halved. In the example of Fig. 9.5 only the gap intervals Y_5 and X_3 are additionally scaled because intervals Y_2 and X_6 are also projections of nodes A and C and can therefore not be scaled. Nodes A and C are then closer together in the final layout of Fig. 9.6.

9.3 Changing the Bounds of a Node

The boundaries of an existing node can, from a user's point of view, be changed by two different editing operations: the node is moved (i.e., its coordinates change) or its size is changed (i.e., the width and/or height and possibly the location change). However, internally we do not have to distinguish between these two operations as they both require the same assistance from the zoom algorithm. The operation of changing the bounds of a node can be described as a combination between the two previously discussed editing operations: The node is removed from its actual location with its current size and inserted at a new location with a new size. Therefore, the layout adjustment that has to be done when the bounds of an existing node change can be realized by combining the operations to remove and insert a node into one single operation. The major problem with this approach is that one single editing operation can result in multiple adjustments of the layout and therefore negatively influence the user's mental map (cf. Section 5.3) and the stability of the layout (cf. Section 5.4). Additionally to this constantly changing layout, it can become difficult to calculate a new final layout if the second adjustment (after the reinsertion of the node) is based on a previous adjustment (the removal of the node).

9.4 Compacting Nodes

In our current tool implementation, we employ automatic layout adjustments only in situations where the moved or resized node overlaps with an existing node or lies partially outside the boundaries of its parent. We do not change the position or size of existing nodes if no overlaps occur. This approach avoids situations in which a very small change of the position or size of a node can have big impacts on the layout that cannot be understood by the user. As a consequence, we currently do not automatically compact the layout if the movement of a child results in additional white space even though this could easily be achieved. For example, if a node is too close to the boundaries of its parent the size of the parent is increased to provide the required space. If the node is then moved away from the parent's border (i.e., to the inside of the parent), the size of the parent is not reduced automatically. Thus the white space at the border of the parent node grows. Automatically reducing the size of the parent node as soon as additional white space becomes available can soon result in a multitude of automatic layout adjustments so that the whole layout starts to oscillate. Thus we do not support an automatic layout contraction during the movement of child nodes (but of course during the removal of them). Instead, we

provide an additional layout operation to compact a node (recursively) by scaling the length of the gap intervals down to a predefined value.

CHAPTER 10

Discussion of the Zoom Algorithm

The zoom algorithm and its extensions to generate different views from a single integrated model and to actively support model editing presented in the last three chapters are now closely examined in this chapter. The techniques are first discussed with respect to the desired properties that where presented in Chapter 5. The second part of this chapter encompasses a more general criteria that has to be discussed for an algorithm, its runtime and space complexity.

10.1 Properties of the Zoom Algorithm

The properties that a zoom algorithm should have so that it can be used for visualizing and editing hierarchical models where presented in Chapter 5. Each of these properties is now discussed in detail for our zoom algorithm.

Compact Layout

The zoom algorithm compacts the layout by scaling the length of the intervals the zoomed-out node is projected to. Two independent scale factors for the horizontal and vertical intervals are calculated from the current and new size of the node so that the amount the diagram shrinks in horizontal and vertical direction can be different. How much an interval can be scaled down depends also on the other nodes that are projected to it. Therefore, it is possible that the zoom algorithm cannot adjust the layout of the model significantly if a node that “lies in the shadow”

of another node is zoomed-out. Node A is shadowed by node B if $proj_x(A) \cap proj_x(B) \neq \emptyset$ or $proj_y(A) \cap proj_y(B) \neq \emptyset$.

For example, zooming-out nodes B and C in Fig. 10.1a) results in a lot of free space between the two nodes in Fig. 10.1b) because they are both shadowed by node A. Nodes A and B are both projected to the vertical interval Y_3 so that $proj_y(A) \cap proj_y(B) = \{Y_3\}$ while C and A are both projected to Y_5. While nodes B and C shrink because they are zoomed-out, intervals Y_3 and Y_5 cannot be scaled down because the size of node A does not change. The height of the zoom holes of nodes B and C (the gray shaded area around B and C in Fig. 10.1b)) is bigger than the height of the two nodes which results in the empty space between them. Unfortunately, there exists no general solution for this problem as it is a manifestation of the trade-off between maintaining the relative position between nodes B and C on one side and the relative position between the two nodes and node A on the other side. This conflict cannot be resolved easily. However, nodes B and C are moved closer together as soon as node A shrinks too (e.g., when it is also zoomed-out) because Y_3 and Y_5 are then no longer blocked by a bigger node.

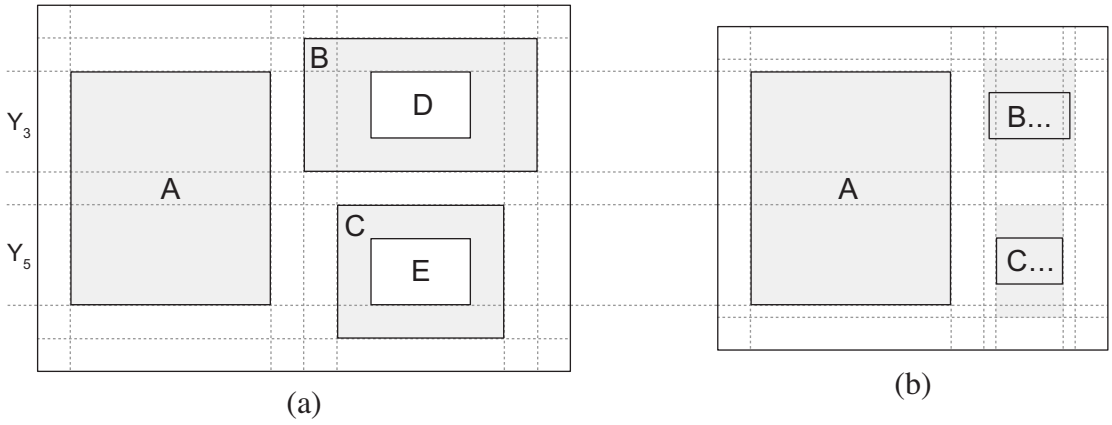


Figure 10.1: Nodes B and C are shadowed by node A

The shadowing of a node can result in situations where only the size of the zoomed node but not the position of the siblings changes when a node is zoomed. If a node is completely shadowed by other nodes both in vertical and horizontal direction, none of the horizontal and vertical intervals it is projected to can be scaled so that the position of all sibling nodes and the size of the parent node remains the same. Node A is completely shadowed both in horizontal and vertical direction by nodes B and C if $proj_x(A) \subseteq proj_x(B) \wedge proj_y(A) \subseteq proj_y(C)$.

Disjoint Nodes

The underlying interval structure of our zoom algorithm and the concept of scaling interval sizes make overlaps between nodes impossible. The horizontal and vertical intervals are stored and adjusted independently and the order within both of these sets is always maintained by the zoom

algorithm. Fig. 10.2 is used to illustrate that a zoom operation does never result in overlaps between nodes which did not overlap before the operation took place. Let's assume that the size of node C is reduced because the node is zoomed-out. In principle, a zoom operation can produce overlaps between the zoomed node and one (or multiple) of its siblings or among two (or more) sibling nodes. An overlap between the zoomed node and one of its siblings is impossible with our zoom algorithm because only the intervals the zoomed node is projected to are scaled. For example, only the vertical node interval Y_4 (and of course the horizontal intervals X_3, X_4 and X_5) but not the vertical gap interval Y_3 that keeps nodes A and B apart from the zoomed node C is scaled down. Node C can never overlap with nodes A and B as long as the ordering of the vertical intervals Y_2, Y_3 and Y_4 is preserved. The same holds for any zoom-in operation. The ordering of the horizontal and vertical intervals is also the crucial point in avoiding overlaps between sibling nodes. The zoom algorithm has to prevent that nodes A and B in Fig. 10.2 are moved too close together so that they overlap. An overlap between these two nodes is not possible as long as the order of the horizontal intervals X_3, X_4 and X_5 is maintained. This is always the case as the zoom algorithm scales interval X_4 (and even maintains a minimal length so that nodes A and B are kept away by a predefined distance) but never removes any of the intervals or changes their order.

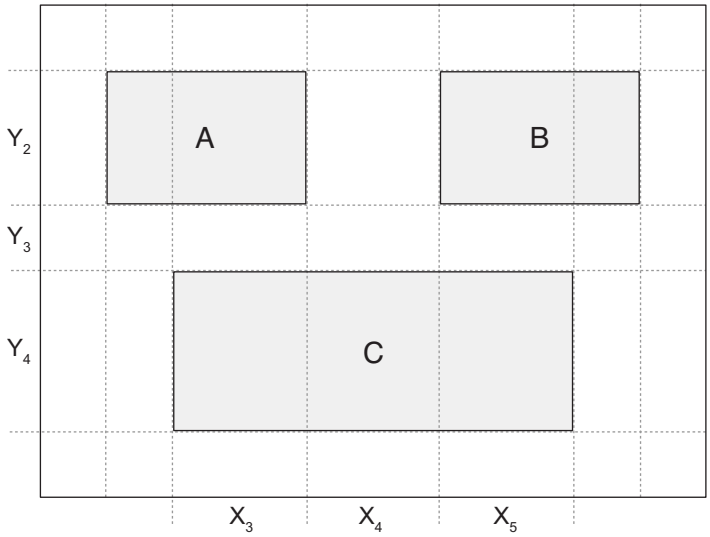


Figure 10.2: Nodes A, B and C are always disjoint after a zoom operation

The property that two nodes A and B do not overlap is true if the two following predicates hold:

$$proj_x(A) \cap proj_x(B) \neq \emptyset \Rightarrow proj_y(A) \cap proj_y(B) = \emptyset \tag{10.1}$$

$$proj_y(A) \cap proj_y(B) \neq \emptyset \Rightarrow proj_x(A) \cap proj_x(B) = \emptyset \tag{10.2}$$

Predicate (10.1) states that the two nodes cannot share a vertical node interval (i.e., both are projected to the same vertical interval) if they already share a horizontal interval. The second predicate (10.2) makes sure that two nodes cannot share a horizontal interval if they already share a vertical interval. For example, nodes A and B in Fig. 10.2 share the vertical interval Y_2 (i.e., $proj_y(A) \cap proj_y(B) = \{Y_2\}$) and cannot share a horizontal interval to fulfill predicate (10.2). It can easily be seen from Fig. 10.2 that they can only share a vertical *and* a horizontal interval if they overlap at least partially. The zoom algorithm makes sure that (10.1) and (10.2) still hold after a zoom operation, if they did both hold initially (as they have to because the initial layout would otherwise already contain overlaps), because it does neither change the order of the intervals nor the intervals the nodes are projected to (i.e., $proj_x$ and $proj_y$ are still the same for all nodes after the zoom operation). Because the zoom algorithm is also used for the adaption of the layout after an editing operation, it can even make sure that a newly inserted or moved node does not overlap with any existing node.

Preservation of the Mental Map

The *orthogonal ordering* between nodes is preserved if the horizontal and vertical order of their center points is maintained (cf. Section 5.3). Our zoom algorithm preserves the horizontal and vertical ordering of the intervals and therefore also the one of the boundaries of the zoom holes (as the boundaries of the zoom holes are defined by the boundaries of the intervals). However, that does not necessarily mean that the orthogonal ordering of the nodes themselves is also maintained because their horizontal and vertical order depends on how they are positioned in their zoom hole (if the zoom hole is bigger than the node).

Fig. 10.3 shows an example. The vertical ordering of the nodes of Fig. 10.3a) is B, C, A (the vertical position of the nodes' centers is indicated by the dotted black lines Y_A, Y_B and Y_C). Fig. 10.3b) shows the situation after node A has been zoomed-out. The vertical ordering is now B, A, C. The vertical position of the centers of nodes A and C is inverted because node A is centered inside its zoom hole (the gray shaded area around node A in Fig. 10.3b)). In principle, it is possible to preserve the orthogonal ordering of the nodes' centers by placing the nodes in a different way inside the zoom holes because the ordering of the vertical intervals is maintained. If the relative position of the center of node A inside its zoom hole is preserved from Fig. 10.3a) to Fig. 10.3b), the vertical ordering of nodes A, B and C is also preserved (as node A is now located at the bottom of its zoom hole).

However, other criteria can also have an influence on how the node should be placed inside its zoom hole. For example, it may be important that nodes A and B overlap vertically so that a link between them can still be represented by a straight horizontal line after node A has been zoomed. The two nodes can no longer be connected by a straight horizontal line if node A is placed at the bottom of its zoom hole in Fig. 10.3b) to preserve the relative position of its center in the zoom hole. Storey and Müller [1995] argue that this *straightness of links* and the orthogonality of links parallel to the X- and Y-axes is an important part of the mental map that should be preserved.

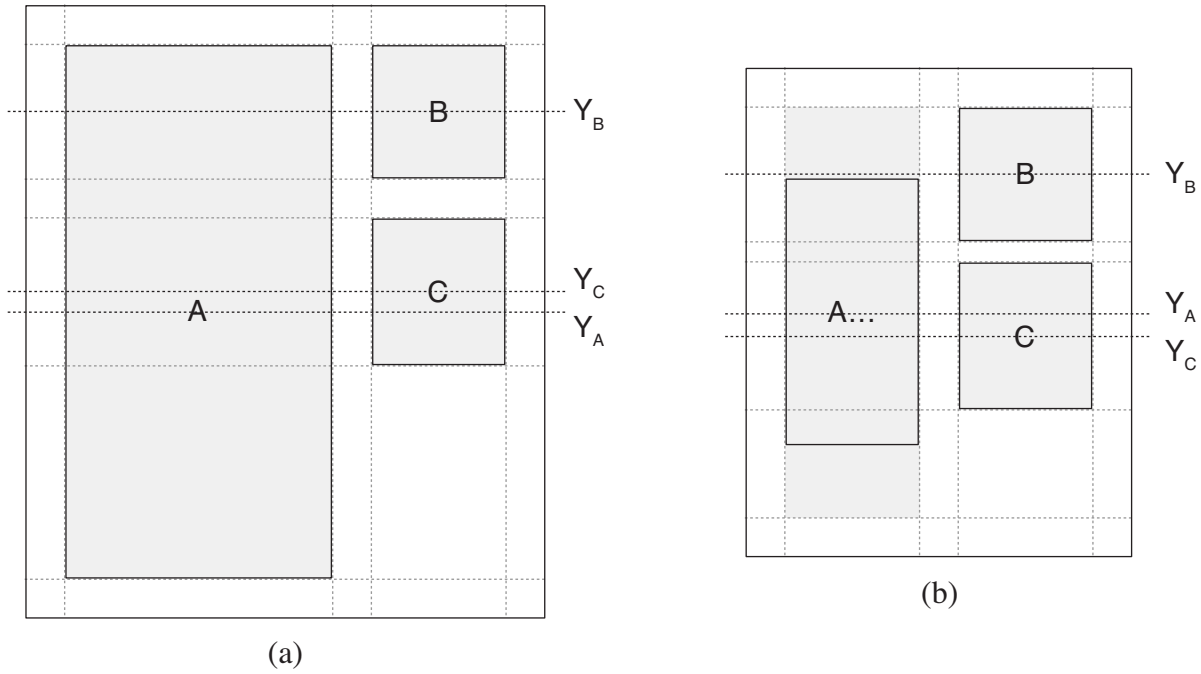
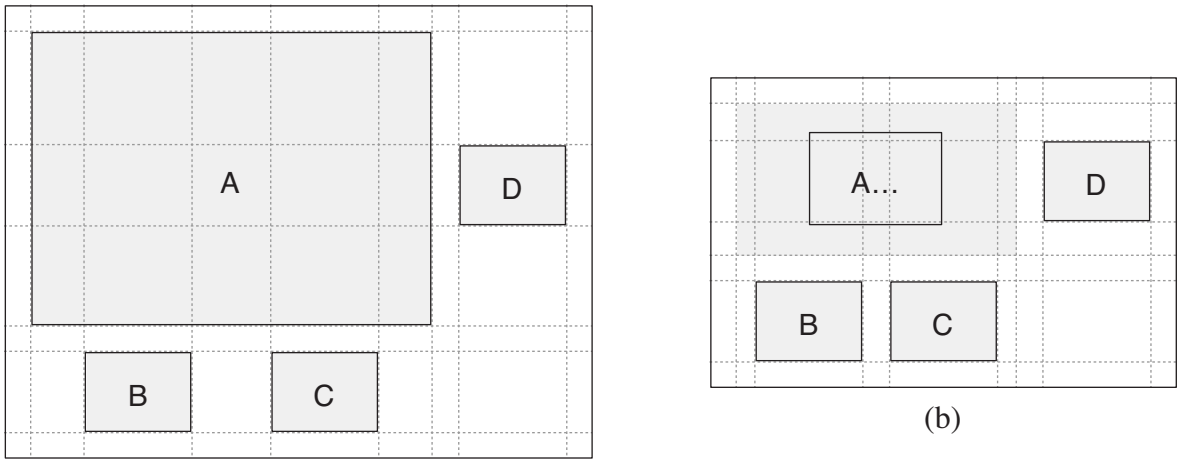


Figure 10.3: The vertical orthogonal ordering between nodes A and D is not preserved

Maintaining the *proximity relations* between nodes when one of them is zoomed-out seems to be an easy task as the whole layout is compacted and the nodes are therefore moved closer together anyway. However, two characteristics of the proximity relations make their preservation a challenge also in the case of a zoom-out operation: First, proximity relations do not only exist between the zoomed node and its siblings but also among these siblings so that there is often a trade-off in deciding which of them should be maintained to what degree. And second, preserving the proximity relations does not only mean keeping close nodes together but also maintaining a certain distance between nodes that where intentionally far apart in the initial layout. For example, consider the situation of Fig. 10.4 where node A is zoomed-out.

Nodes B and C should still be close to node A after the zoom algorithm's layout adjustments. However, it may be of importance too (as part of the secondary notation) that there is a considerable distance between nodes C and D. Thus, the zoom algorithm has to find a compromise so that all proximity relations in the diagram are preserved at least to some degree. The concept of the zoom holes that can become bigger than the node they represent and the property that the ordering of the intervals is maintained makes sure that the basic proximity relations between the zoom holes are always maintained. The proximity relations between the zoom hole of node A, represented by the gray shaded area around node A in Fig. 10.4b), and the zoom holes of the other nodes are still the same as they where in the initial situation of Fig. 10.4a). The proximity relations between the nodes themselves are not exactly the same anymore because they cannot all be maintained without destroying some other properties of the mental map. Maintaining the



(a) (b)

Figure 10.4: Maintain the proximity relations when node A is zoomed-out

horizontal distance between nodes A and D means that nodes D and C overlap horizontally in the final layout which means that the ordering of the horizontal intervals is not the same anymore. Zooming-in node A of Fig. 10.4b) again preserves the proximity relations as it results in the initial layout of Fig. 10.4a). In contrast to the orthogonal ordering and proximity relations, the formal *topology preservation* criterion defined by Misue et al. [1995] cannot be investigated without taking the links of the diagram into account (as the faces of a graph are defined by the links).

Layout Stability

The two most important properties of our zoom algorithm to guarantee the stability of the layout regardless of the order of the zoom operations are the following: (i) the order and number of intervals does not change during a zoom operation and (ii) the scaling is not done on the actual length of an interval but on the real length that is stored for the node whose size changes. The stable order and number of the intervals guarantees that each zoom hole is still projected to the same horizontal and vertical intervals after a zoom operation. The scaling of the real length instead of the actual length makes sure that the length is set to the correct value even if it currently does not have the length it should have after a preceding inverse zoom operation on the same node. The zoom algorithm does the scaling for each node (or more precisely each zoom hole) that is projected to an interval separately and sets the interval's actual length to the minimum of these values. Therefore, two zoom holes that are projected to the same interval cannot influence each other directly as the scaling is done on different values, namely their real length.

For example, consider the situation in Fig. 10.5, where a sequence of zoom operations on nodes A and B results finally again in the initial layout. We consider only the horizontal intervals for

this discussion because nodes A and B overlap in horizontal direction only. Zooming-out node A in Fig. 10.5a) results in the layout of Fig. 10.5b). A's zoom hole is projected to the horizontal intervals X_2 and X_3 (i.e., $proj_x(A) = \{X_2, X_3\}$). Thus, the zoom algorithm should scale these two intervals when the size of node A is reduced. However, interval X_3 cannot be scaled because the zoom hole of node B is also projected to this interval. The zoom algorithm takes care of this restriction by not directly scaling the length of the interval but only the real length of A's zoom hole. This real length is stored separately for each interval the zoom hole is projected to.

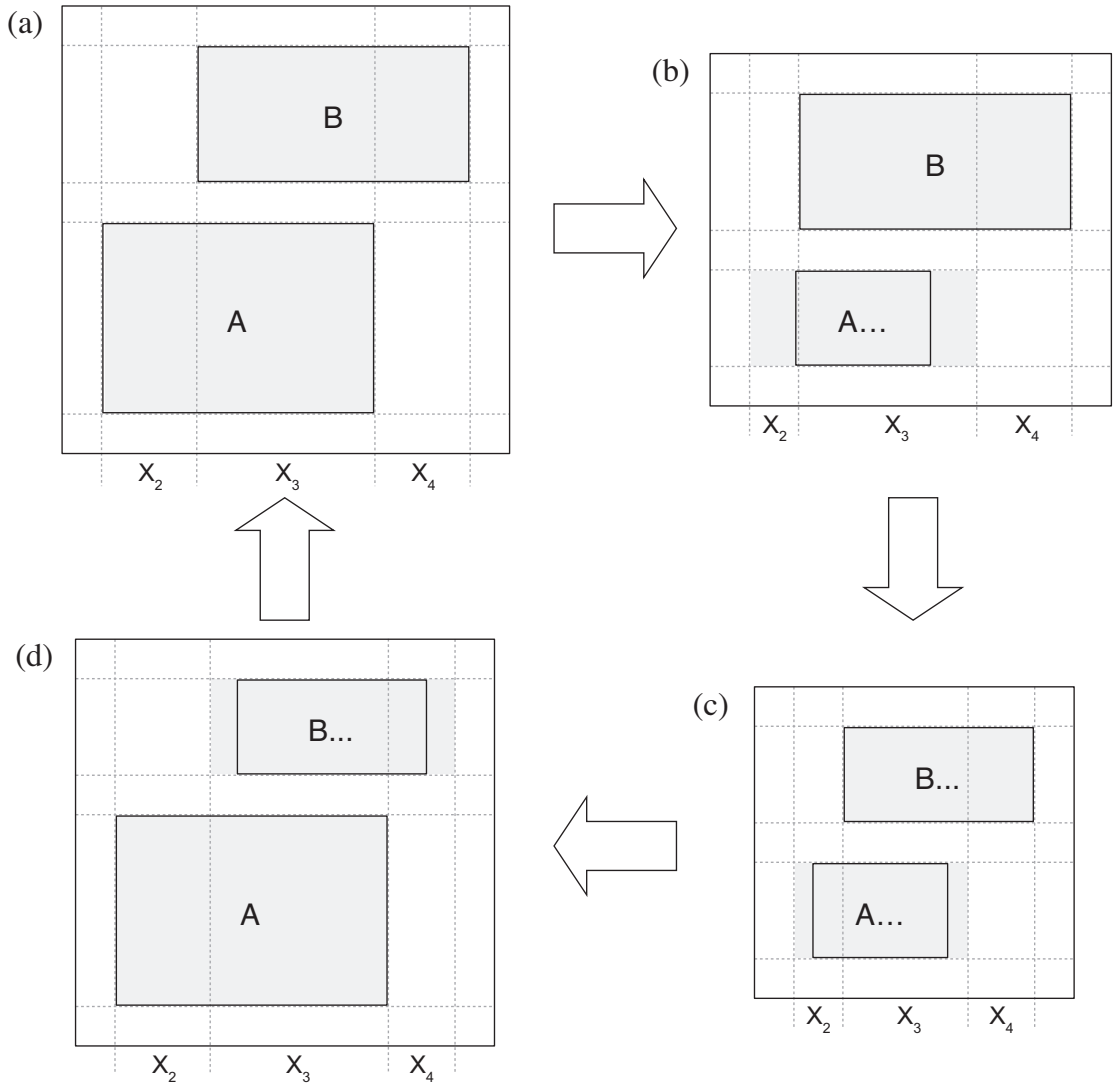


Figure 10.5: Maintain the stability of the layout irrespective of the order of zoom operations

The width of A is reduced by the factor 0.5 so that the real length for the zoom hole of node A is set to $0.5 * len(X_2)$ and $0.5 * len(X_3)$ for the intervals X_2 and X_3. The actual length of the interval is then set to the the maximal real length of all the zoom holes that are projected to the

interval. In the example of Fig. 10.5 this is for X_2 the real length of A's zoom hole and for X_3 the real length of B's zoom hole. The same steps are applied when node B is zoomed-out in Fig. 10.5b) which results in Fig. 10.5c). The real length of node B for X_3 is still bigger than the one of node A because the width of B is scaled by the bigger scale factor 0.7 . The scaling of the intervals is done on the real length of A's zoom hole for the two intervals X_2 and X_3 when node A is again zoomed-in in Fig. 10.5c) which guarantees that A's size is set back to its initial value. The real length of node B for interval X_2 is still stored in the corresponding zoom hole in the situation of Fig. 10.5d) so that zooming-in B does indeed result in the initial layout of Fig. 10.5a).

The situation is more complicated for editing operations. A careful comparison between the layout adjustments in Sections 9.1 and 9.2 reveals that a remove operation does not exactly undo the insertion of a node (i.e., the layout that results after the insertion and a following removal of the same node does not necessarily result in the original layout). The reason behind this is that the zoom algorithm tries to minimize the influence on the layout when a new node is inserted and therefore inserts the node with the maximal size it can have without an overlap with an existing node and then expands it to the intended size (cf. Section 9.1). In contrast, the remove operation removes the whole bounds of the node from the diagram and not just the part that has been additionally added during the expansion step of the insertion. Thus, the zoom algorithm often removes more space from the diagram when a node is removed than it inserted when the same node has been added. However, the opposite situation can also occur: The additional space that is for example added around the node to make sure that there is a minimal gap between the nodes is not removed if the layout is adjusted while the node is deleted. The situation of Fig. 10.6 illustrates the problem where the zoom algorithm removes more space than it has added previously.

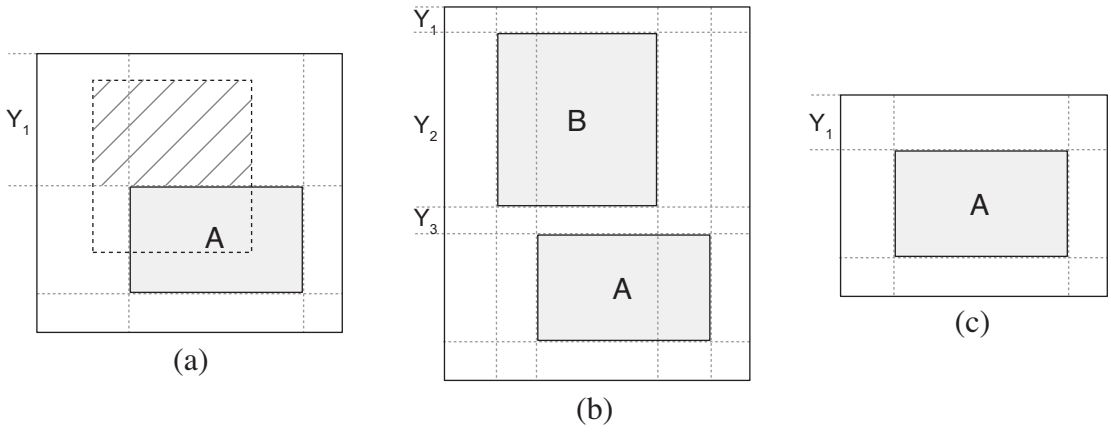


Figure 10.6: Relation between the insert and remove operation

Node B is inserted with the bounds shown by the dotted rectangle in Fig. 10.6a) (an additional border is added so that there is a minimal gap between the existing node A and the new node B). The node is inserted with the size represented by the hatched rectangle. Interval Y_1 is split

at the top boundary of the new node B and the resulting interval Y_2 is scaled so that the whole node fits in. Fig. 10.6b) shows the resulting situation after node B has been inserted. The zoom algorithm now reduces the length of interval Y_2 of Fig. 10.6b) to 1 when node B is removed from the diagram. The length of the first vertical interval Y_1 in the final layout of Fig. 10.6c) is then equal to the sum of the length of intervals Y_1 and Y_3 in Fig. 10.6b) (plus 1 from the scaled interval Y_2). It can easily be seen that the final situation in Fig. 10.6c) differs from the initial situation of Fig. 10.6a). The bounds a node has been inserted with (i.e., the bounds it had before the expansion phase of the zoom algorithm) have to be stored for a later removal if the remove operation has to produce exactly the initial layout.

The main problem of editing operations that change the bounds of a node (cf. Section 9.3) concerning the stability is that the zoom hole is reset to the current size of the node when a node is removed and then reinserted. A zoom hole that is bigger than its node holds the information that is required to preserve the stability of the layout (see Sections 7.2.1 and 7.2.2 for a detailed explanation). Because these informations are lost if a zoom hole is reset, the zoom algorithm can no longer guarantee that the initial layout is restored if a zoomed-out node is moved or resized and then zoomed-in again. That becomes a problem if small changes on the size or position of nodes suddenly result in significant layout changes after a subsequent zoom operation. It is usually not an issue for situations in which the user changes the layout himself significantly because he then understands why the layout cannot be preserved in a subsequent zoom operation. Fig. 10.7 illustrates a situation where a slight movement of node B while it is zoomed-out compromises the stability of the layout. Fig. 10.7a) shows the initial situation with node B zoomed in (its children are omitted since they are not relevant for the discussion). Node B is then zoomed-out, as shown in Fig. 10.7b) and moved slightly away from node A towards the bottom of the figure in Fig. 10.7c). Subsequently, zooming-in the moved node B results in the layout of Fig. 10.7d) which differs from the initial layout of Fig. 10.7a).

The reconstruction of the initial layout from the situation of Fig. 10.7b) relies on the fact that the zoom hole of B (the gray shaded area around B) is bigger than the node itself. The information that the left boundary of node B is to the left of node A and its right boundary to the right of node A is stored in the zoom hole. The size of the zoom hole is reset when the zoomed-out node B is moved in Fig. 10.7c) because the node is first removed from the interval structure and then inserted again. The width of the new zoom hole of node B is now smaller than the width of the zoom hole of node A. The zoom algorithm tries to preserve this information by stretching the zoom hole of node A further in Fig. 10.7d) so that it horizontally still covers the zoom hole of node B. The size of the root node in the final layout is bigger than its size in the initial layout even though the geometric relations between nodes A and B are preserved (together with the slight downwards movement of node B). The same problem can occur if a new node is inserted into the area that is occupied by the zoom hole which is bigger than its node.

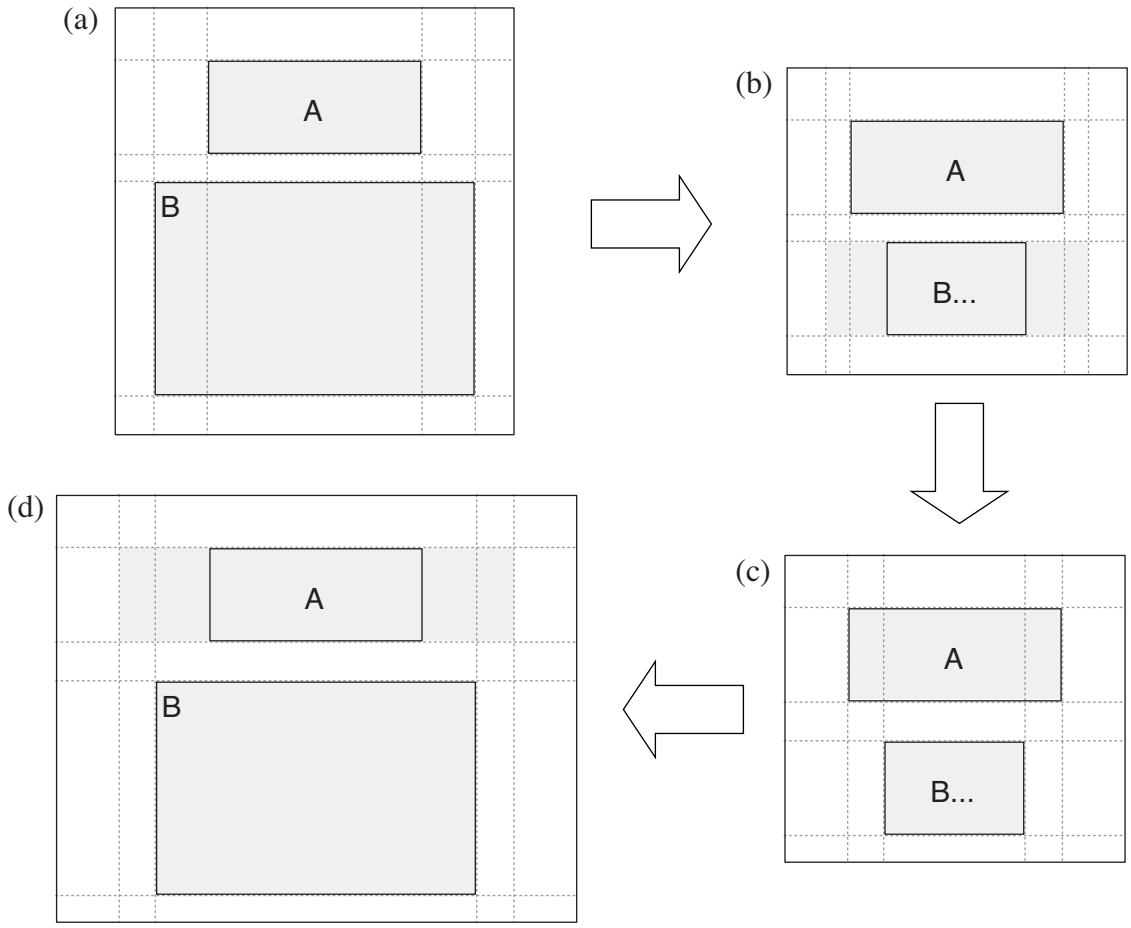


Figure 10.7: Layout is no longer stable after an editing operation

Permit Editing Operations

The presented zoom algorithm does not only permit editing operations but actively supports them as described in Chapter 9. However, this property comes not for free: Editing a diagram while some of its nodes are zoomed-out can make it impossible to guarantee the stability all the time and preserve all properties of the mental map. The layout is not stable anymore if one or multiple zoom holes are reset during editing operations as described in the last section.

Runtime

The evaluation whether our fisheye zooming techniques makes it possible to perform zooming operations with no remarkable delay has to be done by examining the runtime complexity of the algorithm which is done in Section 10.2.

Multiple Focal Points

Each zoom operation is handled separately by our zoom algorithm so that the nodes can be in any zoom state (zoomed-in or zoomed-out) at any time. Having multiple nodes zoomed-in at the same time shows the diagram with multiple focal points. The user can freely decide on the number and location of these focal points while the zoom algorithm guarantees the stability of the layout regardless of the number of focal points and the order they have been applied in.

Smooth Transitions

An animation of the transition between two layouts can be realized independently of the zoom algorithm by linearly interpolating the nodes' size and position between the initial and final value. Animating the layout changes does not only help in enhancing the continuity for zoom operations but for any layout adjustment which is done by the zoom algorithm such as the generation of different views (cf. Chapter 8) and the smart editing (cf. Chapter 9).

Small Interaction Overhead

The interaction that is needed for a zoom operation is kept as simple as possible in our fisheye zooming approach: The user just has to click on a zoomed-out node to zoom into it while clicking on a zoomed-out node results in the abstracted view of the node. Clicking on a node "opens" or "closes" the node depending on its current state. The size of a node is always calculated automatically and is either defined by the space that is required by the node's children or a predefined minimal size. A zoom operation does only change the size of the zoomed node and recursively the size of its direct and indirect parents but never the size of one of its siblings. The sibling nodes are only moved to provide the required space or remove empty space from the diagram. This mode of interaction differs in its simplicity from other approaches such as SHriMP/Creole (cf. Section 6.2) where the user first has to open a node by clicking on it and then scale its size freely by holding the mouse button.

Minimal Node Size

A minimal node size is only needed for nodes whose children are all hidden because the node is either zoomed-out or all its children are filtered out in a specific view. The size of the zoomed node's siblings cannot fall below the minimal size as it is not changed at all. That is different from other approaches such as SHriMP (cf. Section 6.2) and the Continuous Zoom (cf. Section 6.4) where the size of all nodes changes if one node is zoomed so that the diagram still fills the screen completely after a zoom operation. Our zoom algorithm does not pose any restrictions on the size a zoomed-out node should have so that it can be set to any predefined or calculated value (see Section 7.2.2 for a discussion of different ways to calculate the size of zoomed-out nodes).

10.2 Algorithmic Complexity

Our fisheye zooming technique consists of two parts: the interval structure as the underlying data structure and the algorithms that work on this structure. The interval structure is constructed only once and then adjusted if a zoom operation is applied. Therefore, we can examine the complexity of the interval structure construction and the zoom algorithm independently. The interval structure is constructed incrementally by inserting each node when it is added to the diagram as described in Section 7.1. The algorithm to insert a node into the structure has linear complexity with respect to the number of intervals because it iterates (independently) over the horizontal and vertical intervals. The same holds for the other operations that work on the interval structure (such as removing a node or adjusting its size) so that the number of intervals that a structure can maximally have defines the runtime complexity of the algorithms that work on it. An interval structure with n nodes can have at most $2n+1$ horizontal and $2n+1$ vertical intervals as illustrated by Fig. 10.8. Fig. 10.8a) shows the number of intervals for one node. There are at most three vertical and three horizontal intervals for a structure with one node. There may be less intervals if one boundary of the node overlaps with one boundary of the interval structure. Each additional node adds at most[1] two vertical and two horizontal intervals. This is shown in Fig. 10.8b) for a second node.

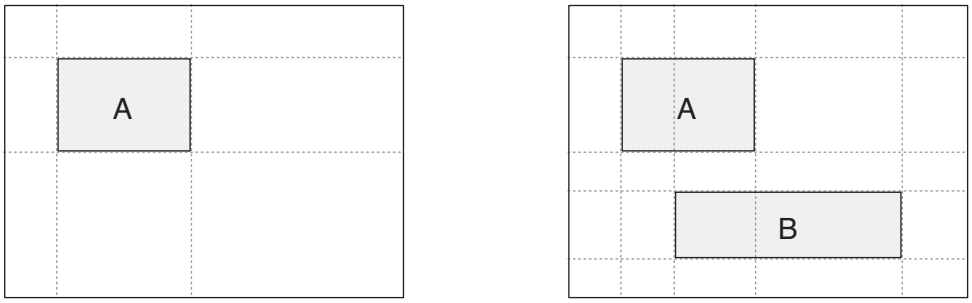


Figure 10.8: Number of intervals in the interval structure

For zoom operations, the complexity of adjusting the interval structure dominates the complexity of constructing the structures because the structures are only adjusted and not newly constructed if a node is zoomed. The runtime complexity of our zoom algorithm depends linearly on the number of intervals in the interval structure because the algorithm has to scale the node intervals of the zoomed node to adjust the layout. Thus, the runtime and space complexity are linear with respect to the number of nodes. The linear runtime complexity guarantees that our zoom algorithm can be used in an interactive environment. The space complexity is important because some informations (i.e., current and real length of the interval) about the structure have to be stored.

[1]The number of additional intervals is lower if the boundaries of the nodes are aligned.

Part III

Line Routing

CHAPTER 11

Lines in Graphical Models

The lines that are used to represent relationships between nodes in a diagram are often as important as the nodes themselves because the primary task of node-link diagrams is the visualization of relations. The effort needed to recognize relations is largely determined by the drawing of the lines in the diagram. Crossings and overlaps between lines and nodes and among different lines have a negative impact on the readability and understandability of a diagram. Such overlaps occur frequently in the modeling domain where nodes can, in contrast to abstract graphs where nodes take up little or no space, have a considerable size. However, manually arranging the lines so that such overlaps do not occur burdens a lot of additional work on the user who should be released from these tedious layouting tasks. Therefore, an automatic line router that directs line around nodes while trying to keep the lines as simple as possible is needed.

This chapter sets the stage for such an automatic line router by first discussing the specific problems of the routing task in hierarchical models (Section 11.1) and in layouts that are constantly changed and updated by zoom and/or editing operations (Section 11.2). These specific characteristics are taken up with a more general description of the line routing problem by discussing a set of requirements that a routing algorithm must meet (Section 11.3).

11.1 Lines in Hierarchical Models

The hierarchical relations in a diagram can either be depicted explicitly by links, as for example the composition and aggregation relations in UML's class diagrams [OMG, 2005], or implicitly

by the nested box notation (cf. Section 3.3.1). The nested box notation represents parent-child relationships visually by the child being completely contained within the parent node. This reduces the number of links because some of the relations are represented implicitly. However, representing hierarchical decompositions by nested boxes also entails some specific problems concerning the lines in the diagram.

11.1.1 Identifying Potential Obstacles

Lines in a hierarchical diagram often connect nodes located in different branches of the hierarchy tree[1]. Therefore, it is not always obvious which nodes are potential obstacles for a line. For example, the bold nodes in Fig. 11.1a) are the potential obstacles for the line connecting nodes F and J. These nodes lie on different levels of the hierarchy. For example, nodes G and D are both potential obstacles even though node G is one hierarchical level below node D as can easily be seen in the tree representation of Fig. 11.1b). The potential obstacles of a line have to be computed by traversing the hierarchy tree from node p the line is completely contained in (which is node A in Fig. 11.1) to both the source and target node of the line. Thereby, all the nodes on each level which are not direct or indirect ancestor of either the source or target node are collected. Once the potential obstacles have been identified, the hierarchical line routing problem becomes an instance of the usual "flat" line routing problem.

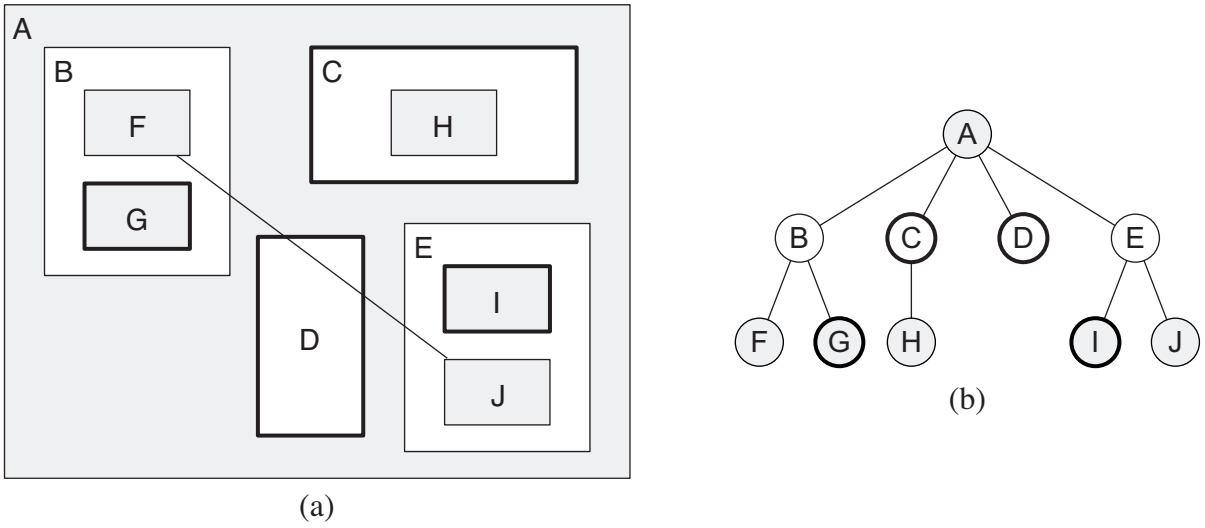


Figure 11.1: Nodes C, D, G and I are potential obstacles for the link between F and J

[1]The hierarchy of the model can also be a forest (i.e., a set of trees with no single root node). In this case it can easily be transformed into a tree by adding a "virtual" root node as parent for the existing root nodes of the individual trees.

11.1.2 Abstract Relationships

An additional problem that occurs only in hierarchically nested and never in flat models is that links must reflect the hierarchical structure of the model: A relationship between two nodes implies relationships on all hierarchical levels above these nodes. Otherwise, the information that there exists a relationship between two nodes is lost if at least one of the direct or indirect parents of one of these nodes is zoomed-out. In ADORA these implied higher-level relationships are called *abstract relationships* [Glinz, 2002; Glinz et al., 2002]. Moody [2006] uses the term "boundary relationships" and Harel [1987] calls them "stubbed entrance arrows" in his statechart notation. For example, consider the situation of Fig. 11.2 which shows again the model of Fig. 11.1a) but this time with node B zoomed-out. The relationship between nodes F and J cannot be shown by a link because node F is not visible when its parent B has been zoomed-out. The fact that there exists a relationship between node J and one of the (currently invisible) children of B is represented by an abstract relationship, drawn as a bold line, between nodes B and J. The concept of abstract links guarantees that the model cannot become inconsistent concerning the information flow between the objects represented as nodes [Glinz et al., 2002]. However, for the line routing task it has the consequence that the lines have to be newly routed after each zoom operation because existing lines can disappear and new lines emerge.

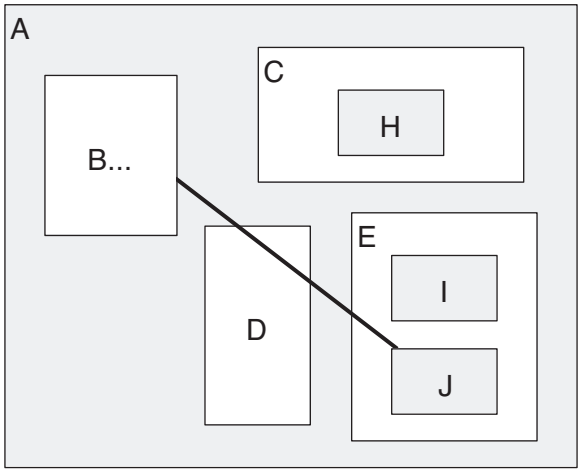


Figure 11.2: Abstract relationship between nodes B and J after B has been zoomed-out

Both concepts, the calculation of the potential obstacles and abstract relationships, are only needed if the links that can occur in a diagram are not restricted by the hierarchical relationship between their source and target node. No hierarchical restriction means that two nodes can be connected by a link regardless of their position in the hierarchy. An explicit port concept, as it is for example employed in UML's Composite Structure or Component Diagrams [OMG, 2005], constrains the connectivity so that links are only allowed between sibling nodes (i.e., nodes that are on the same hierarchical level and have the same parent). This restriction reduces the complexity of the line routing task significantly because of the following reasons: (i) the potential

obstacles do not have to be calculated because only the sibling nodes of the connected nodes are candidates, (ii) the concept of abstract relationships is not necessary as the links never cross the boundaries of a node and (iii) the line routing algorithm can exploit the partitioning of the model through the decomposition structure because a line never leaves the parent of the two nodes it connects (which reduces the number of affected nodes).

11.2 Lines in a Dynamic Layout

The dynamic layout resulting from the zoom operations and the creation of different views intensifies the need for an automatic line router because the lines have to be adjusted to the new situation after each layout change. It makes additional demands on the routing algorithm because the lines have to be routed in real-time in order to avoid that the runtime performance of the zoom algorithm (cf. Section 5.6) is adversely affected by the adjustment of the lines that has to be done after each zoom operation. As fisheye navigation requires the generation of a new diagram layout in every navigation step, automatic and fast (i.e., real-time) line routing is essential and a manual adjustment of all lines is just not an option.

There are two possible ways of how to adjust a line if the layout changes due to a zoom operation or the switch to a different view: (i) Treat the bending points in the lines as nodes and let the zoom algorithm handle them (as for example proposed by Bartram et al. [1995] and Berner [2002]) or (ii) newly route the affected lines after each layout operation. The problem with the first solution is that the layout changes that occur during a zoom operation can have a big impact on a line even tough the overall layout is not changed dramatically. New bend points may become necessary to avoid that a line suddenly passes through a node and existing bend points can become obsolete because the line can now be routed in a more direct manner. Fig. 11.3 shows an example where the line between nodes A and C can be routed directly without an additional bend after node B has been zoomed out in Fig. 11.3b) even though the overall shape of the diagram has not changed significantly.

In addition to the zoom algorithm, the layout can also be changed by the user directly when he edits the diagram. The tedious task of good line routing has be done automatically too when a diagram is edited so that the modeler can concentrate on his primary goal of creating or modifying a model. Diagrams are in the modeling domain usually created manually and incrementally by a human user and not automatically from an existing abstract representation. Hence, we cannot compute diagram layouts (including the lines) from scratch as done in graph drawing (cf. Section 2.3.1) or the software visualizations in the reengineering field. Instead, we must be able to adapt a given layout incrementally, preserving the secondary notation as far as possible. A dynamic layout (i.e., zoom operations, view generation, interactive editing) results in some extra demands on the line routing algorithm but it is also the biggest advantage a tool has over a simple paper based modeling process. Thus, each algorithm that automates at least parts of this dynamic layout creation is highly valuable.

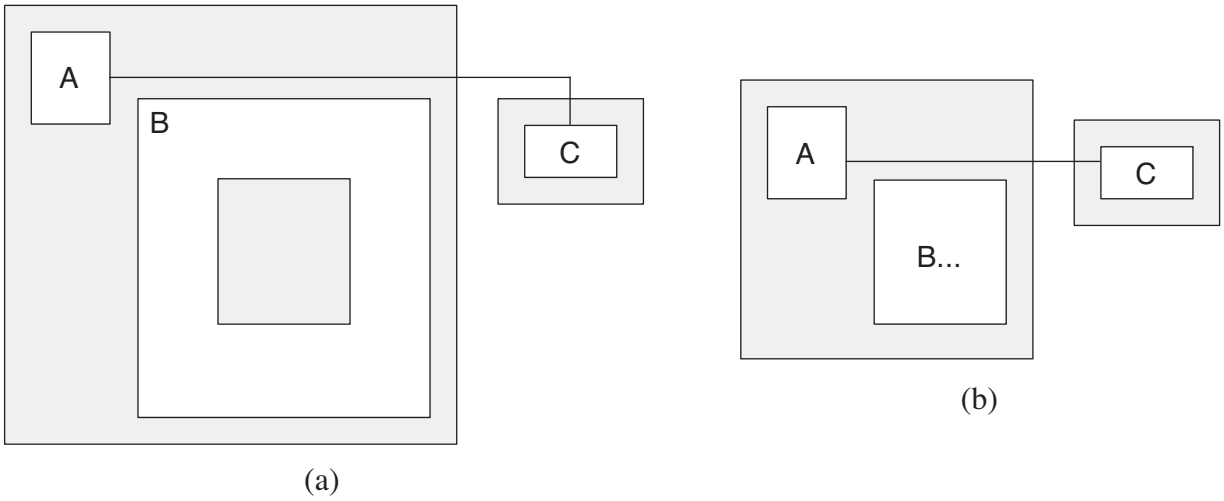


Figure 11.3: Direct line between nodes A and C after node B has been zoomed-out

11.3 Desired Properties of a Line Routing Algorithm

The last section of this chapter briefly discusses the properties a line routing algorithm should have so that it can be used for hierarchical dynamical models in a similar way as it was done for the zoom algorithm in Chapter 5. The lines in a graphical model or diagram have gained much less attention in the literature and are often treated as a second rate citizen beside the nodes. That is a little bit surprising because the central task of node-link diagrams is to depict relations. Most of the work stems from the graph drawing field where the edges or links are often the most important among the factors that govern automatic graph drawing algorithms (cf. Section 2.3.1).

11.3.1 Bypass Nodes

The main rationale behind a line routing algorithm is to avoid that a line crosses freely through nodes that lie between its source and target. The problems of such overlaps between nodes and lines are that (i) the readability is reduced because the line may cross the label of the node, (ii) it becomes harder to follow a line if it crosses the border of a node (which is in fact represented by a set of lines) and (iii) understanding the internals of a node in a hierarchical model is hard if lines that do not have anything to do with these internals are drawn (partially) inside the node. Therefore, lines must not pass through nodes to avoid such node/link crossings. Overlaps between links and nodes become only a problem if the nodes take up a lot of space because of a text label or nested child nodes in a hierarchical diagram. This is usually not the case in the graph drawing field where the algorithms are designed for abstract graphs whose nodes take up little or no space [Misue et al., 1995].

11.3.2 Minimize the Length of a Link

Most automatic graph drawing algorithms (cf. Section 2.3.1) try to avoid crossings between links and nodes while placing connected nodes close together. Thus, minimizing the length becomes an integral part of the layout and is often not even mentioned as a specific requirement anymore. While layout algorithms can rearrange the nodes to reduce the length of the links this is not an option for manually arranged diagrams. However, the links should still be as short as possible. This ensures that the user finds a link at the place he first looks for which is usually the area between the connected nodes.

11.3.3 Minimize Link Crossings

Since diagrams are a means of communication between different people with different backgrounds, it is crucial that the diagrams present information clearly. An empirical study by [Purchase et al., 2002] has revealed that minimizing the number of crossings between lines is the most important aesthetic that should be considered while trying to draw a diagram automatically (cf. Section 2.4). Therefore, reducing the number of line crossings is one of the driving forces behind most graph drawing algorithms (cf. Section 2.3.1). They try to place the nodes so that crossings between lines are avoided. A lot of these algorithms work only on planar graphs (i.e., graphs that can be drawn without any edge crossings) so that a graph has to be "planarized" first (by for example deleting some of the edges). The situation is different in the modeling domain where the position of the nodes is determined by the user, as an important part of the secondary notation, and must not be changed by the line routing algorithm. Thus, the only option for the line routing algorithm is to find another path through the nodes that results in less crossings. The disadvantage of this procedure is the resulting trade-off between finding the shortest path and minimizing the number of line crossings that results because a line usually has to take a detour to avoid a crossing which in turn increases the length of the line.

11.3.4 Routing Styles

In contrast to other domains where lines or links are used (e.g., circuit design), diagram aesthetics play an important role in the modeling domain. Battista et al. [1994] distinguish between three different graphic standards or styles that have been proposed for the representation of graphs in a plane. The nodes are usually represented by graphical symbols such as circles or boxes and each link is a simple open curve between the symbols associated with the start and end node. A drawing in which each link is represented by a polygonal chain is a *polyline* drawing. Two special cases of polyline drawings are *straight-line* drawings that map each link onto a straight-line segment and *orthogonal* drawings that map each line onto a chain of horizontal and vertical segments. The straight-line graphic standard is commonly adopted in the graph theory field while technical diagrams such as Entity Relationship or UML diagrams are usually drawn according to

the orthogonal graphic standard. Polyline drawings can also be modified to produce curved lines (i.e., splines).

The lines of a diagram have to be easy to follow and add clear meaning to the diagram. Ware et al. [2002] investigated the influence of different graph aesthetics on finding the shortest path in a set of graphs. Their primary focus was to show the importance of good continuity of the lines (i.e., keeping lines with multiple segments as straight as possible). The basic idea behind this is to exploit the Gestalt principle of continuity (which states that we are more likely to construct visual entities out of visual elements that are smooth and continuous, rather than ones that contain abrupt changes in direction) by using smooth and continuous lines as links [Ware, 2004, page 191]. The results of their experiments showed that the continuity is, after the obvious length of the path itself, the most important factor in perceiving the shortest path. Therefore, Ware et al. [2002] state that it may be worth to accept an occasional crossing in a graph layout if it reduces the "bendiness" of a line. Additionally, they argue that splines should be preferred to orthogonal lines because their changes in direction are smoother and enhance continuity.

However, the semantic domain of the graph or diagram affects which aesthetic criteria need to be emphasized [Purchase et al., 2002] and it is impossible to state that some of the aesthetic criteria are universally valid. The experiment by Purchase et al. [2002] revealed that non-orthogonal drawings were preferred to orthogonal drawings because of the increased number of bends in the orthogonal lines. But orthogonality was then in fact preferred in the semantic domain of UML class and collaboration diagrams. One of the reasons for this may be that in most areas that use diagrams for visualizing information a preferred style or consensus about how a line should look like has emerged (e.g., orthogonal lines in circuit diagrams or UML diagrams). People are used to that style and become confused if this convention is violated. Additionally, the results indicate that the overall layout has an impact on how a line should look like. Diagrams that consists of mostly rectangular figures seem to be candidates for orthogonal lines while straight lines seem to be more suited for rooted trees such as family trees or organization charts.

It is possible to take advantage of the fact that none of the proposed line routing styles is objectively superior to any of the others. Different line routing styles can be combined in one diagram to route different kinds of links with different styles. This technique simplifies the distinction of different kinds or types of relationships but makes the line routing task harder because the routing algorithm has to support different routing styles.

11.3.5 Mental Map and Secondary Notation of a Line

Which characteristics of a line are part of its secondary notation and influence the user's mental map is much less examined than in the case of the nodes' characteristics (cf. Section 5.3). North [1996] argues that the preservation of the mental map concerning the nodes is much more crucial than the preservation of the lines' mental map. The rationale behind this is that nodes are landmarks that a user can learn and return to in a diagram, while lines are usually traced on

the fly to discover relations or connections[2]. If lines are indeed secondary in the preservation of the mental map, it becomes advantageous to adjust the lines aggressively (e.g., by newly routing them after each layout adjustment) to improve the layout, while moving nodes more conservatively. Improving the layout means in this context fulfilling some of the general aesthetics such as minimizing the number of line crossings or bends in a line.

However, Fig. 11.4 shows that the line routing may have an important effect on the overall look and thus on the mental map of a node-link diagram: Diagrams a) and b) look different even though the underlying structure (i.e., the underlying graph) and even the location and size of the nodes are the same. The differences result solely from varying the position of the lines' start and end point on the boundary of their start and end node. Therefore, completely ignoring the lines while considering the mental map of a diagram may result in a significant disorientation of the user. A discussion of metrics to formalize and measure the mental map of a line can be found in Section 5.3.

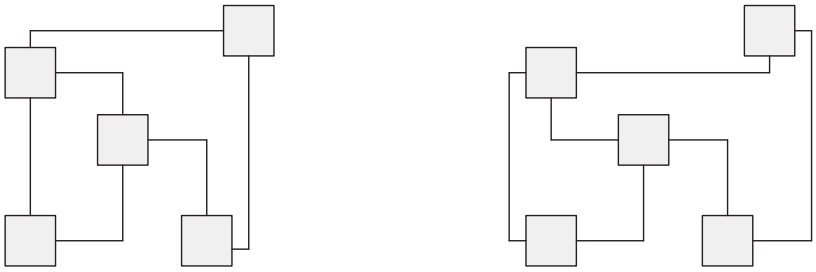
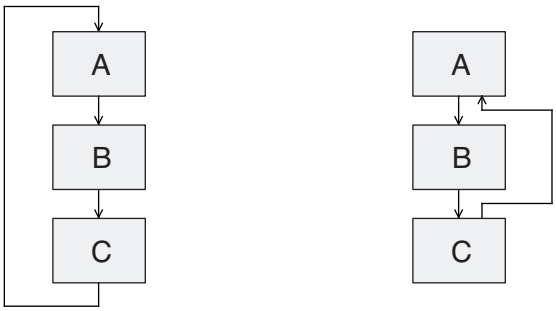


Figure 11.4: Line routing and the mental map [Bridgeman and Tamassia, 2000]

Besides their influence on the formation and preservation of the mental map, some characteristics of a line may, additionally to the aesthetic criteria, have an influence on the readability and understandability of a diagram. The direction of the information flow has emerged as an important aesthetic feature during the experiments of Purchase et al. [2002]. The direction of the information flow is foremost defined by the position of the nodes that are connected. The position of a node is either defined by the user or the zoom algorithm and must not be changed during the line routing. But the direction of information flow can be emphasized by selecting the position of the start and end point on the boundaries of the start and end node accordingly. The example of Fig. 11.5 shows a repeated sequence of steps A, B and C with an information flow from top to bottom. This direction convention is preserved for the line connecting nodes C and A in Fig. 11.5a) as the line leaves node C at the bottom and enters node A at the top even tough C is located below A. As a result, the repeating pattern can be recognized a lot easier than in the syntactically equivalent layout of Fig. 11.5b).

[2]This is also the reason why it is much more common to hide links and not nodes to reduce the complexity of a diagram.



(a) (b)

Figure 11.5: Secondary notation employed in a line

Thus, the routing algorithm has to give off some control to the user so that he can at least partially influence the final shape of the line to encode additional information in its secondary notation. This can for example be achieved by letting the user determine the start and end point of the line. The routing algorithm must then preserve this secondary notation as far as possible if the line is (re)routed after a layout operation. However, the preservation of the secondary notation or mental map and the aesthetic criteria such as a minimal length of the line or a minimal number of line crossings are competitive so that the optimality of one often prevents the optimality of others. For example, the line between nodes C and A in Fig. 11.5a) is longer than strictly necessary because the secondary notation is used to show the direction of the information flow consistently.

11.3.6 Runtime

The runtime complexity of the routing algorithm is an important issue because the lines have to be routed in real time whenever the modeler changes the layout of a model by navigating, creating a view or editing (cf. Section 11.2). Thus, the line routing algorithm must be able to route a large fraction of all lines in a diagram without a remarkable delay. The concrete number of lines that have to be rerouted depends on how many nodes are changed by the layout operation and how the lines are embedded into the hierarchy.

CHAPTER 12

Existing Line Routing Approaches

The problem of routing a line around a set of obstacles occurs in many different domains. Since the detailed requirements and constraints for the routing of a line vary slightly from domain to domain, different techniques have evolved rather separately and isolated from each other. The following sections give a short overview over different techniques from various domains and discuss their applicability to graphical models with respect to the desired properties presented in Section 11.3. The first section covers the routing of lines or edges by automatic graph drawing algorithms. Section 12.2 describes the use of a visibility graph for the routing that plays an important role in computational geometry. Approaches to route wires in the circuit design field are discussed in Section 12.3. Finally, Section 12.4 presents a technique that has been developed especially for incrementally created models of software systems.

12.1 Lines as Part of the Automatic Graph Drawing

In the automatic graph drawing field the routing of lines or edges is an integral part of the automatic graph drawing algorithm (cf. Section 2.3.1). One of the main goals of such an algorithm is in fact to avoid line crossings. The routing of the lines is foremost optimized by placing the nodes in a manner that reduces crossings between lines. Two characteristics of graphical models render the application of this integrated layouting and routing approach impossible. First, the layout is usually done manually and incrementally by the user in order to exploit the secondary notation (cf. Section 2.5) and must not be changed dramatically in case of a small change to preserve the user's mental map. And second, most of the automatic graph drawing algorithms were originally

designed for abstract graphs were nodes take up very little or no space [Misue et al., 1995]. In contrast, the graphical representations of nodes in graphical models of software systems take up a considerable part of space, especially if they employ the nested set notation (cf. Section 3.3.1).

12.2 Routing on the Visibility Graph

The "geometric shortest path problem" in computational geometry has many applications in robotics, geographic information systems and diagram drawing (see e.g. [de Berg et al., 2000] for an overview). Probably the best known approach constructs a visibility graph and computes the shortest path on this graph according to Dijkstra's algorithm [Dijkstra, 1959]. Fig. 12.1 shows an example of a simple visibility graph used to route a line that connects points S and T along the obstacles represented by the gray rectangles. The vertices of the visibility graph (shown by the little circles in Fig. 12.1) are the corners of the obstacles together with the start and end point of the path. There exists an edge between two vertices v and w if the two vertices can "see" each other (i.e., the straight line segment $\overline{vw}$ does not intersect the interior of any obstacle). The shortest path (or more precisely one of the shortest paths since it needs not be unique) can then be found by assigning the Euclidean length of an edge as its weight and then using Dijkstra's shortest path algorithm. The black vertices and edges in Fig. 12.1 constitute the shortest path between points S and T.

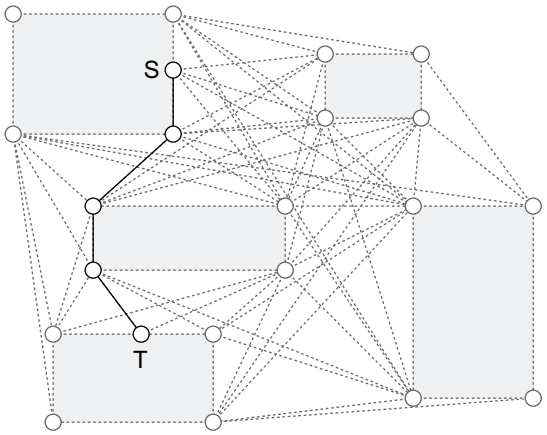


Figure 12.1: Visibility graph

Relying on a visibility graph for the routing of lines in a graphical model yields two problems. First, the avoidance of line crossings cannot be integrated directly into the shortest path calculation on the visibility graph. This is due to the fact that the visibility graph does not contain any informations about existing connections between nodes (i.e., it represents only the relations between the corners of nodes in the original layout). Thus, crossings between lines have to be detected and resolved in an (expensive) second step after the basic routing. And second, the routing is restricted to straight line drawings since the edges in the visibility graph are direct straight

lines. Orthogonal lines that are common in technical drawings or models cannot be deduced in a straightforward manner from such a visibility graph.

12.3 Grid-Based Routing

The circuit design domain has evolved independently from the automatic graph drawing field even though the circuits constitute with their electronic elements as nodes and the wires as edges a graph. This is due to mainly two reasons: First, the layout of circuits has to satisfy technical (e.g., maximal lengths of the wires or the prohibition of wire crossings) instead of aesthetic criteria. Additionally, the prohibition of line crossings implies that the lines cannot be routed independently from each other. Instead, there has to be a specific order in which lines are routed since an already routed line becomes a potential obstacle for subsequent lines. And second, in contrast to the integrated view in the graph drawing field, the layouting of the circuit (i.e., the position of the elements) and the routing of the wires are treated as two separate steps. Furthermore, the number of nodes and links is, especially in the field of highly integrated circuits (VLSI: Very Large Scale Integration), with several millions very high.

The wires in electric and electronic circuits are almost exclusively routed orthogonal. Only some analog circuits employ an angular routing style. Because wire crossings are not allowed, wires are usually not only routed on one plane but on several stacked layers. Therefore, the routing takes no longer place in a 2D space but in a 3D space. However, for the following discussion we restrict ourselves to two dimensions since the presented concepts can easily be extended to three dimensions. Almost all wire routing techniques operate on a uniform grid structure. Because the wires have to be routed on this structure around existing electronic elements that represent obstacles like in a maze, this concept is often called “maze routing”.

12.3.1 Lee’s Algorithm

The first and perhaps best known algorithm for grid-based routing in circuit designs is Lee’s algorithm [Lee, 1961] which is an application of Dijkstra’s breadth-first shortest path search algorithm [Dijkstra, 1959] to a uniform grid. Lee’s algorithm is based on the expansion of a diamond-shaped wave from the source point that continues until the target point is reached. Fig. 12.2 shows an example to find a path connecting grid cells S and T. The gray shaded cells represent obstacles that cannot be crossed by a wire or line. The first and second step of the wave expansion are show in Fig. 12.2a) and b), while c) shows the situation after the eighth iteration with the wave approaching the end cell T. The shortest path can then be found in a second step by going back from the target point to the source point while selecting the neighbored grid cell with the lowest value. This is illustrated by the hatched cells in Fig. 12.2d). The algorithm always finds a solution if one exists, and ensures an optimal solution. The major drawback of this approach is that its space and runtime complexity is $O(mn)$ for a grid with $m*n$ cells which

makes it unsuitable for an interactive use. Several techniques have been proposed to reduce the space and runtime complexity of the algorithm. For example, the approach of Soukup [1978] combines the breadth-first with a depth-first search by looking always into the direction of the target without changing the direction. Only after hitting an obstacle, the algorithm falls back to a breadth-first search. However, this approach sometimes results in suboptimal solutions and can still not reduce the large number of cells which constitute the uniform grid structure.

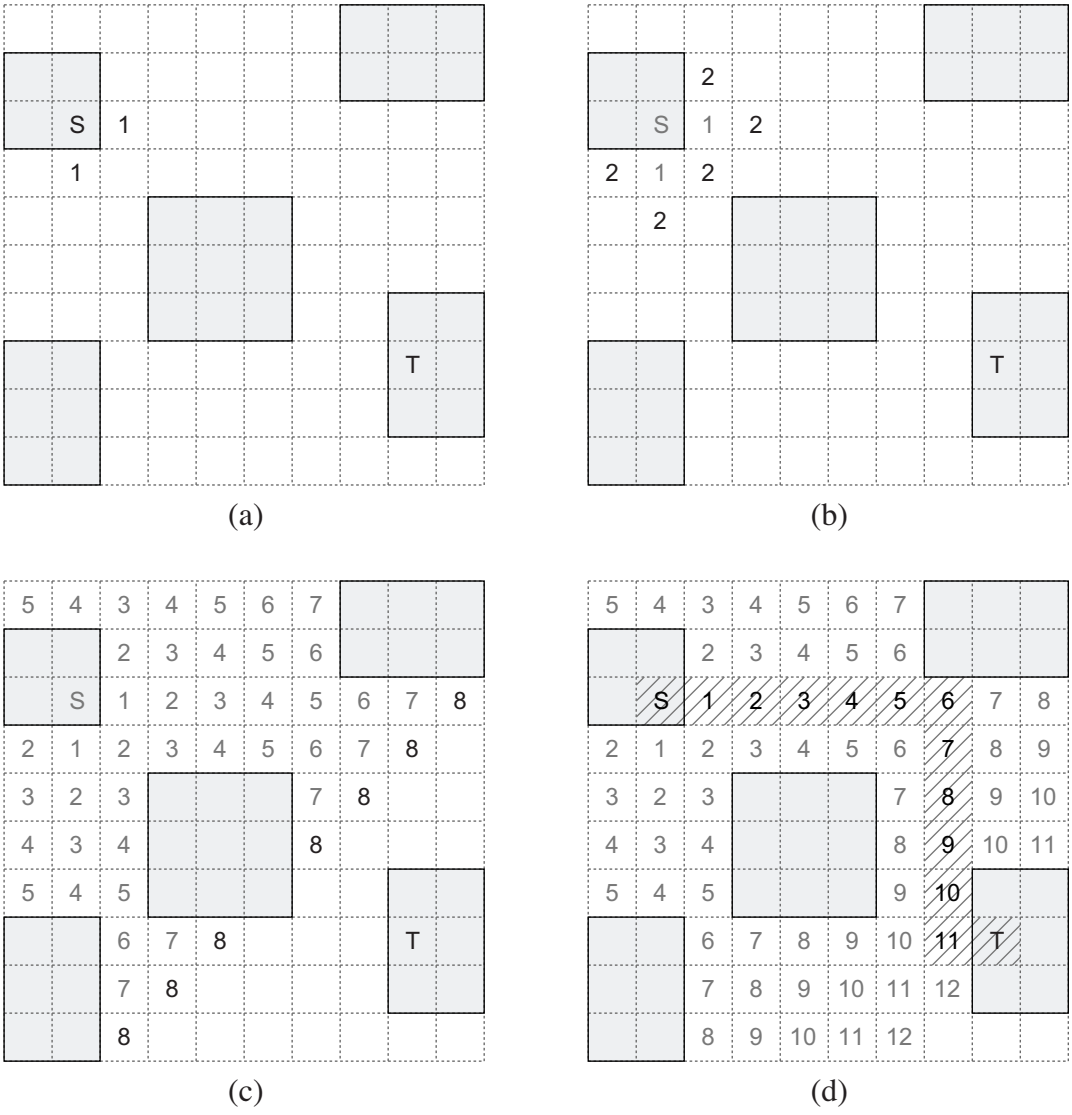


Figure 12.2: Lee's algorithm

12.4 Routing in Incremental Layouts

The concept of Miriyala et al. [1993] explicitly addresses the problem of routing lines in models of software systems that are incrementally constructed by the user by adding nodes and links one after the other. Their approach produces orthogonal lines (since that is the routing style that is most commonly used in graphical models of technical systems) and tries to minimize the number of crossings among lines, the number of bends in lines and their overall length. The concept is very similar to our own algorithm[1] presented in the next chapter. The two main differences are that we use a different underlying data structure (tiles instead of segments) and that we separate the path (or channel) finding from the actual routing of the line. These two characteristics make it possible to employ additional routing styles while the concept of Miriyala et al. is restricted to orthogonal lines.

The data structure that is used by Miriyala et al. is the so called rectangulation. It represents each node as a rectangle and each link as a sequence of horizontal and vertical segments (thus the restriction to orthogonal lines). The reason for this subdivision into rectangular regions is that bends or crossings can only result from a movement from one region to another. The rectangulation is constructed by drawing a so called virtual vertical segment from every critical point (i.e., corner point of a node or bend point in a line) that extends up to the next object that obstructs the path. Fig. 12.3 shows an example:

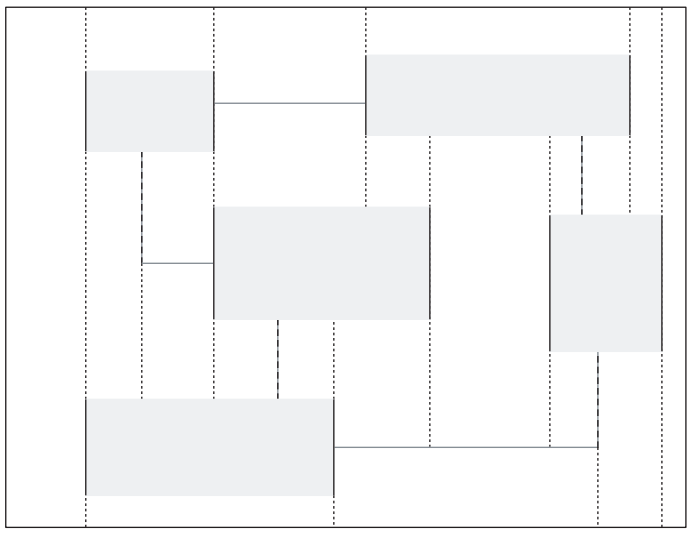


Figure 12.3: Rectangulation

The gray rectangles and lines in Fig. 12.3 represent the original nodes and links. The dashed lines illustrate “edge segments” (i.e., vertical segments of the existing links), the continuous

[1] We developed our algorithm independently since we were not aware of the concept of Miriyala et al. [1993] at that time.

black lines “vertex segments” (i.e., vertical bounds of the nodes) and the dotted lines “virtual segments” which are added to construct the rectangulation. Dijkstra’s shortest path algorithm [Dijkstra, 1959] is then used as a heuristic to find line routings on this structure similar to our own approach as presented in Sections 13.2 and 13.3.

CHAPTER 13

Tile Maze Router

The major problem of Lee's algorithm presented in the last chapter is the large number of cells in the grid structure. The resulting runtime and space complexity can be reduced by applying the basic idea of the algorithm to a sparse structure instead of a uniform grid. This "corner stitching data structure" and some of the basic operations that can be applied to it are presented in Section 13.1. Our line routing approach is divided into two completely decoupled steps: The first step, described in detail in Section 13.2, computes one or multiple paths through which the shortest path has to pass. The second step which computes the line itself (i.e., the bend points in a polyline) is shown in Section 13.3. Extensions of the basic algorithm are presented in Sections 13.4, 13.5 and 13.6. The chapter finishes with a discussion of our algorithm in Section 13.7. A preliminary version of the algorithm has already been presented in [Reinhard, 2004; Reinhard et al., 2006].

13.1 Data Structure

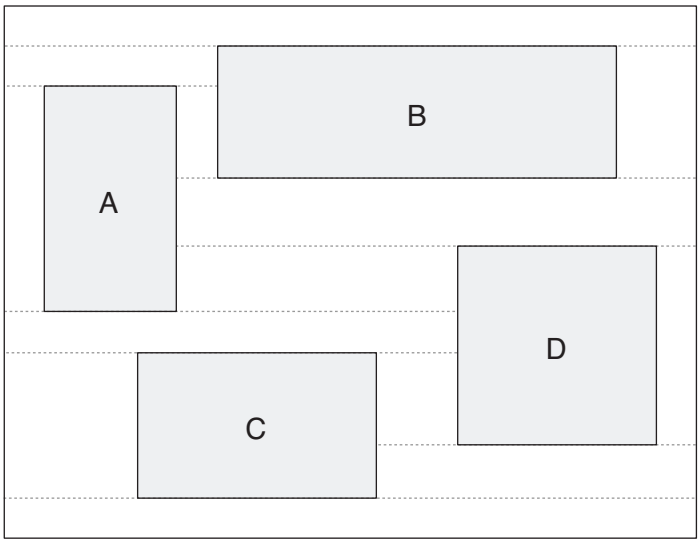
Instead of a uniform grid we use the corner stitching structure [Ousterhout, 1984] as the underlying data structure for the line routing. The corner stitching structure has originally been developed as an efficient storage mechanism for VLSI layout systems and has two important features:

- All space, whether occupied by a node or empty, is explicitly represented in the structure. This explicit representation of the empty space makes it possible to provide fast geometri-

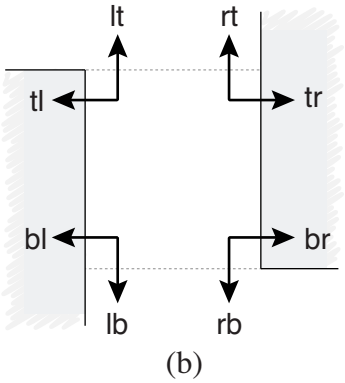
cal algorithms to locate the space that is available for the routing.

- The space is divided into rectangular areas that are stitched together at their corners like a patchwork quilt. These corner stitches allow easy modifications of the structure and lead to efficient implementations of a variety of geometric operations.

The corner stitching structure for the four nodes of Fig. 13.1a) is represented by the dashed lines. The space is divided into a mosaic with rectangular tiles of two types: *space tiles* and *node tiles*. Tiles must be rectangles with sides parallel to the X and Y axes. The only constraint is that nodes must not overlap. This has to be guaranteed by other techniques such as the ones described in Chapters 7 and 9. The space tiles are organized as maximal horizontal strips: no space tile ever has another space tile immediately to its right or left. However, there can be another space tile directly above or below a space tile. This organization ensures that there is one and only one decomposition of the empty space into space tiles for each arrangement of node tiles. The maximal horizontal strip representation is crucial for the space and time complexity of the geometric operations. Fig. 13.1b) shows in detail how the tiles are linked by a set of pointers, called corner stitches, at their corners. Ousterhout [1984] uses only four pointers (two at the bottom-left and two and the top-right corner) instead of eight. We use the additional four pointers for our extension to minimize line crossings (cf. Section 13.4).



(a)



(b)

Figure 13.1: Corner stitching structure

The advantage of a corner stitching structure with maximal horizontal strips over a uniform grid structure is its linear space complexity: while the number of cells in a uniform grid is determined by the size and the resolution of the grid, the maximum number of space tiles in the corner stitching structure only depends on the number of nodes. In a diagram with n nodes, there will never

be more than $3n + 1$ space tiles (see [Ousterhout, 1984] for a proof). Additionally, all operations on the corner stitching structure depend only on informations of their local neighborhood so that the size of the overall diagram has no negative influence on their runtime. However, this clear upper bound can only be achieved by a higher space complexity (i.e., we face an instance of the runtime-space trade-off). The corner stitching structure requires at least three times more memory than a simple linked list since it requires empty space to be represented and uses more neighbor pointers. Fortunately, the space complexity is not a big problem for diagrams as the number of nodes is usually at most several hundred and reaches never the millions of nodes that are no exception in the VLSI domain.

The question that remains is why we have to use an additional data structure for the line routing instead of just reusing the interval structure that is already used by the zoom algorithm (cf. Section 7.1). The problem is that the horizontal and vertical intervals are independent in the interval structure because we want to scale them independently. The intervals do not represent an area of the layout but are in fact projections from the two-dimensional Euclidean space to a one-dimensional space. For the line routing algorithm we need a structure that divides the plane in a set of areas or tiles because we want to use the basic algorithm of Lee [1961] which works on a two-dimensional representation of the space.

Finding the Tile at a Given Location

One of the most common geometric operations in a diagram is to find the node at a given (x, y) location. For example, this operation can be conducted each time the mouse pointer is moved to find the node it currently points to. The simplest but also least efficient solution for this problem is to sequentially iterate over a list containing all nodes in the diagram. This approach has linear runtime with respect to the number of nodes in the diagram. The corner stitching structure permits a much more sophisticated algorithm to find for each point p, represented by its coordinates (x, y), the tile the point lies in. The algorithm can be used to search for both, node and space tiles. It iterates in horizontal and vertical direction over the tiles, starting from any given tile in the structure:

1. The algorithm follows the corner stitches pointing to the top and bottom neighbor of a tile (lt and lb in Fig. 13.1b)), until a tile is reached whose vertical range contains the vertical position of the desired point p.
2. From this tile it moves to the right or left, by following the tr and tl stitches, until a tile is found whose horizontal range contains the horizontal position of the desired point p.
3. This horizontal movement can result in a vertical misalignment (i.e., the tile that was found during the horizontal search in step 2 does not contain the vertical position of the desired point) so that steps 1 and 2 may have to be repeated several times to locate the tile containing p. Whether a point is located in a tile can easily be found out by comparing the boundaries of the tile and the position of the point.

Fig. 13.2 illustrates how the tile containing point p is located in a tile structure starting from tile E which is located in the top left corner of the diagram. In the first step, the algorithm iterates vertically over tiles E, F, G, H and I until the tile that contains the vertical position of p is reached, which is in this case tile I. The horizontal iteration over tiles I, C and J ends in tile J because this tile contains the horizontal position of p. However, this horizontal movement has now introduced a vertical misalignment (i.e., tile J does not contain p) so that the vertical iteration of step 1 has to be repeated starting from tile J to reach tile K which contains point p.

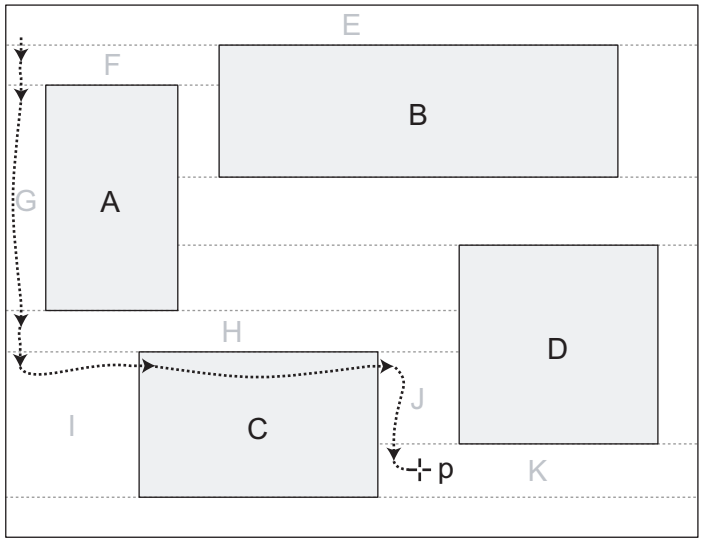


Figure 13.2: Locating the tile that contains point p

In the worst case, the algorithm has to iterate over all tiles in the structure (such a pathological case occurs, for example, if all nodes are in one column or row). Fortunately, the runtime in the average case is much better [Ousterhout, 1984]: $\sqrt{n}$ nodes have to be visited in a structure with a total of n space and node tiles.

Neighbor Finding

Another common search operation is to find all tiles that touch one side of a given tile. The corner stitches make this operation particularly simple. For example, finding all right neighbors of a given tile T is achieved by first following the tr corner stitch to find the topmost right neighbor. A vertical iteration starting at this neighbor and then following the lb corner stitches until the tile the br stitch of T points to is reached gives all the tiles that touch the left side of T. The runtime of a neighbor finding operation is linear with respect to the number of neighbors.

Construction of the Corner Stitching Structure

At the beginning, the corner stitching structure consists of one single space tile that occupies the whole layout. Node tiles are then incrementally inserted for each node in the diagram. Because we use the structure only for the line routing and not any other layouting task, we construct the structure after the layout (i.e., position and size of the nodes) has been established. Therefore, we can assume a correct layout and do not have to check whether the node that is inserted into the structure overlaps with an existing node (even though such a test can easily be performed with the corner stitching structure). The algorithm to insert a tile into an existing structure is illustrated by means of Fig. 13.3.

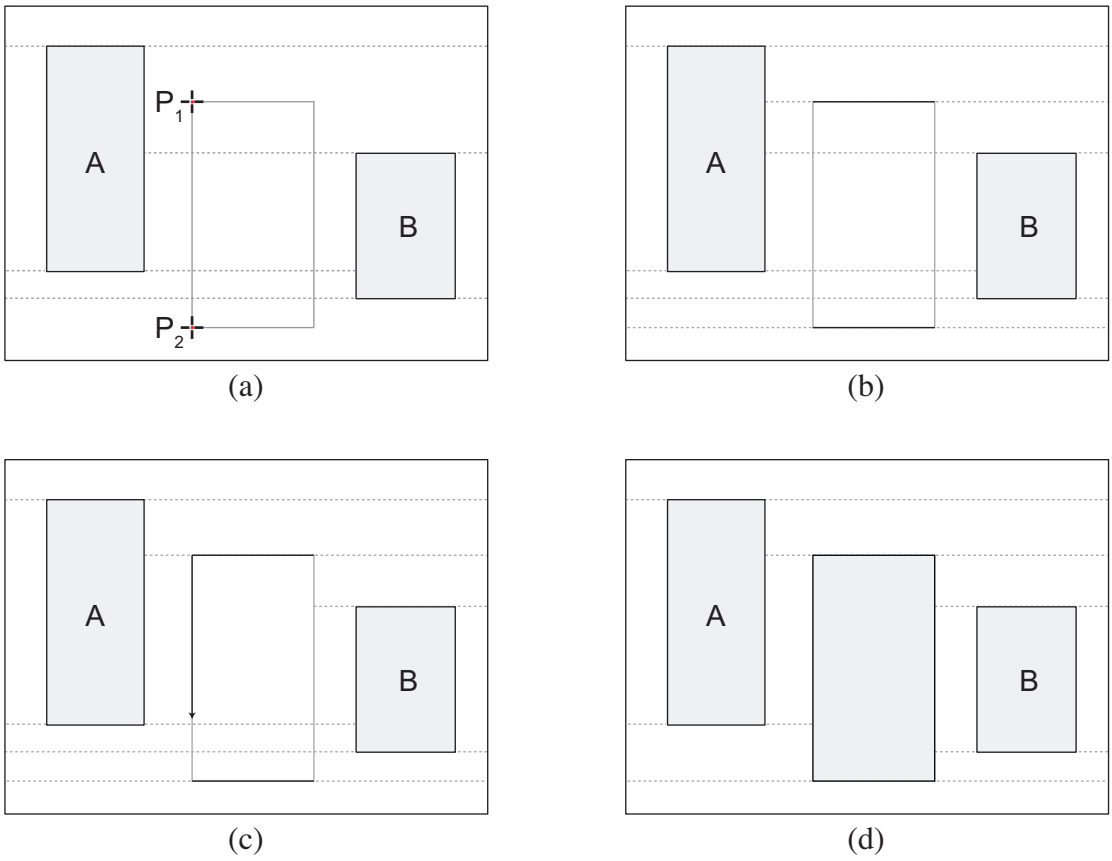


Figure 13.3: Inserting a tile into the corner stitching structure

1. The algorithm to find a tile at a given location is used to search for the space tile that contains the top-left border of the node to be inserted (P_1 in Fig. 13.3a)). The organization of the corner stitching structure guarantees that the whole top boundary of the node lies in one single tile.

2. The found space tile is split along the line defined by the top boundary of the node to be inserted as shown in Fig. 13.3b). This implies that the corner stitches of the involved tiles and its neighbors are updated.

3. The algorithm to locate a tile at a give point is now again used to find the space tile that contains the bottom-left corner of the node that should be inserted (P_2 in Fig. 13.3a)).

4. This bottom space tile is, in analogy to step 2, split and the corner stitches are adjusted so that they point again to the correct neighbors.

5. The algorithm now iterates vertically over the tiles starting from the top tile of the new node until the bottom tile is reached. During this iteration each tile has to be split vertically in two new tiles: one to the left and one to the right of the new node. Space tiles are whenever possible vertically merged into one tile. Thereby, the corner stitches of the involved tiles and their neighbors are adjusted.

The runtime of the algorithm depends on the effort that is required for the splitting and merging of the tiles. This effort is determined by the number of tiles that have to be visited (i.e., the neighborhood of the new node). The merging and splitting can in the average case be done in constant runtime as long as the new tile has about the same size as the tiles around it (see [Ousterhout, 1984] for a detailed discussion).

On-the-fly Construction

Because the construction of the corner stitching structure requires some effort and even tough the runtime becomes only a problem in really large diagrams, the question arises of how to store the structure after its initial construction. In principle it would be possible to construct a corner stitching structure for each node in the hierarchy (by inserting the children of the node one after the other into the structure) and then store it. That is in fact the same strategy that was used for the interval structure that is used for the fisheye zoom (cf. Section 7.1). However, the problem with this approach is that the corner stitching structures are not bound to a specific node but rather to a specific line because the potential obstacles for a line (i.e., the nodes that have to be inserted into the structure) are defined by the hierarchical position of the line's start and end node (cf. Section 11.1). Storing a structure for each line instead of node is possible but requires that it is updated after each layout change. Because the removal and reinsertion or movement of a tile can become quite complex and a new structure can be constructed with a justifiable effort, it is easiest to construct a new structure each time a line is routed.

13.2 Channel Finding

The first step of our routing algorithm computes the path(s) of space tiles through which the line has to pass by applying the fundamental wave expansion idea of Lee's algorithm to the corner stitching structure instead of a uniform grid structure. During the expansion phase the algorithm computes a distance value for the space tiles of the structure. Due to the non-uniform tile size, it is not possible to use the distance from the source point to each of the tiles as distance value, because there may be multiple different values for one tile. Therefore, we use a combination of the source distance and the distance to the target point which are both measured in the Manhattan distance[1]. Furthermore, for each tile the algorithm computes the point P inside the tile where the distances are actually measured. For the actual search, an ordered data structure (e.g., a heap or a priority queue) denoted as Ω is used. The algorithm consists of the following steps:

1. The corner stitching structure is constructed and tile T_{start} that contains the source point s and tile T_{end} that contains the target point t are identified.

2. Point P for T_{start} is set to the source point s, the source distance of T_{start} to 0 and the distance of T_{start} to the Manhattan distance between P and the target point t. T_{start} is then inserted into Ω.

3. As long as Ω contains tiles: Tile T with the lowest distance value is removed from Ω. If this tile is T_{end}, a path has been found. Otherwise, point P_{next} (i.e., the point inside T_{next} closest to the point P of the current tile T) is calculated for each neighboring space tile T_{next}. For each of these neighboring tiles two distance values are calculated: The *source distance* σ is calculated by adding the Manhattan distance between point P and point P_{next} to the source distance of T. The *distance* δ is calculated by adding the Manhattan distance between P_{next} and the target point t to σ. Equations 13.1 and 13.2 show the calculation of the source distance σ and the distance δ for the tile T_{next} relative to the tile T, whereas $\lambda(P_1, P_2)$ denotes the Manhattan distance between the points P_1 and P_2:

$$\sigma(T_{next}) = \sigma(T) + \lambda(P, P_{next}) \tag{13.1}$$

$$\delta(T_{next}) = \sigma(T_{next}) + \lambda(P_{next}, t) \tag{13.2}$$

   The calculated distance value for T_{next} is compared to the previously calculated value for T_{next} (if there is one). If it is lower, the distance δ and source distance σ values of T_{next} are updated. Tile T_{next} is then inserted into Ω.

4. The sequence(s) of space tiles that constitute the shortest path can be found, in analogy to Lee's and Dijkstra's algorithm, by moving back from T_{end} to T_{start} while repeatedly

[1]The Manhattan distance, also known as the L_1-distance, between two points P and Q is defined as the sum of the lengths of the projections of the line segments onto the coordinate axes: $\lambda(P, Q) = |x_P - x_Q| + |y_P - y_Q|$

selecting a neighbored tile that has the same distance value δ. If multiple neighbors have the same value, there exist multiple solutions and the current tile is a branch.

Fig. 13.4 illustrates the proceeding of the routing algorithm by means of a simple example. As an initial simplification, we assume that the start and end point of the line have already been projected from their nodes to one of the space tiles around the node by some means (cf. Section 13.5). Thus, the start point s lies now in the space tile T_1 below node A and the end point t in space tile T_6 above node D.

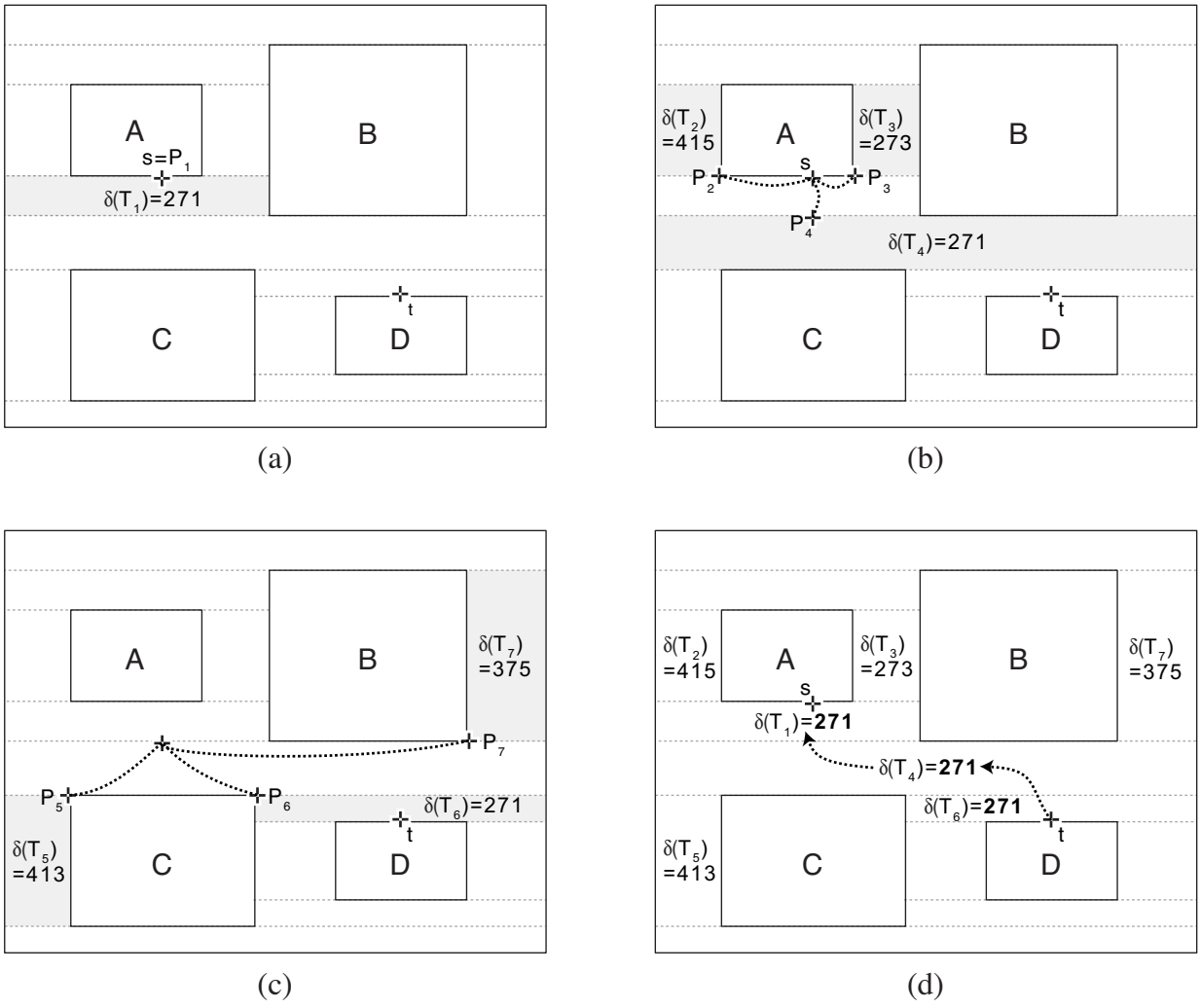


Figure 13.4: Finding a path of space tiles from s to t

Point P_1 of tile T_1 is set to the start point s, the source distance $\sigma(T_1)$ to 0 (because the distance between s and P_1 is zero as they are at the same position) and the distance $\delta(T_1)$ to the Manhattan distance between points P and t which is 271. T_1 is then inserted into the ordered data structure

Ω and immediately removed again in the next step because it is the first and only tile that is currently in Ω. The algorithm now calculates the point P_{next} for each of the T_1's neighbors, T_2, T_3 and T_4, as shown by the dotted black lines in Fig. 13.4b). The resulting points are P_2 for T_2, P_3 for T_3 and P_4 for T_4. The next step is to calculate for each of these neighboring tiles the source distance σ and the distance δ. For tile T_4, for example, the source distance is calculated according to formula (13.1) by adding the Manhattan distance between P_1 and P_4 to the source distance $\sigma(T_1)$ of the current tile which results in $\sigma(T_4) = 32$. The distance $\delta(T_4)$ is computed according to formula (13.2) by adding the distance between P_4 and the target point t to the just calculated source distance $\sigma(T_4)$. The Manhattan distance $\lambda(P_4, t)$ is 239 so that the distance for tile T_4 is equal to 271. T_4 has the same distance value as tile T_1 because the increase in the source distance is compensated by the decrease of the distance to the target when P moves away from the source point closer to the target point. That is not the case for tiles that lie on a path that goes into the "wrong direction" (i.e., further away from the target point instead of closer to it) like for example tile T_2 for which both the source distance and the distance to the target increase. The combined distance value and the fact that the tiles are taken from an ordered data structure Ω results in a directed search instead of Lee's undirected search which propagates in all directions. The ordered data structure makes sure that the algorithm does automatically look for an alternative path as soon as it is stuck with the most promising one. Fig. 13.4c) shows the next iteration through step 3 of the algorithm: T_3 has been removed from Ω because it is the tile with the lowest distance and the distance values for its neighbors T_5, T_6 and T_7 are calculated. The path has been found as soon as T_6 is processed in the next iteration because it contains the end point t. The gray shaded rectangles which represent in Fig. 13.4 the tiles for which the distance values are currently calculated show how the wave expands from s towards t. Fig. 13.4d) shows how the sequence of the space tiles T_6, T_4 and T_1 that constitute the shortest path can finally be found by repeatedly selecting a neighbored tile with the same distance value as the current one.

13.3 Bend Points Calculation

The algorithm for finding the space tiles the shortest path has to pass through is completely decoupled from the algorithm that does the actual routing (i.e., computes the coordinates for drawing the line). Hence, different algorithms that produce different styles of lines can be implemented on top of the algorithm described in the last Section.

13.3.1 Orthogonal Lines

As an example, we have implemented an orthogonal line router that produces rectilinear polylines. The algorithm starts with the path returned by the path finding algorithm and iterates over all tiles in the path by repeatedly following the reference to the next tile. The line is routed inside a space tile with respect to the direction the line enters and leaves the tile. In the following, the four possible directions are named by the points of the compass *north*, *south*, *east* and *west*. We

distinguish between three possibilities of how a line can pass through a single space tile:

1. The line crosses the tile vertically, which means that the line enters the tile from north and leaves it to the south or enters it from the south and leaves it to the north as shown in Fig. 13.5a). Fig. 13.5b) shows that it is also possible that the line goes back into the same direction in came from. This direction can either be south or north but never west or east because of the organization of the corner stitching structure as maximal horizontal strips.

2. The line crosses the tile horizontally by entering it from west or east and leaving it into the opposite direction. Because of the organization of the corner stitching structure, this situation can only occur if the respective tile contains both the start and the end point (i.e., the path consists on one single tile as shown in Fig. 13.5c)).

3. The line changes its direction inside the tile. That is the case if the line enters the tile from north or south and leaves it to the west or east or, vice versa, enters from west or east and leaves to north or south. This situation can only occur if the tile in the west or east is either the start or end tile (i.e., the tile that contains the start or end point) as illustrated in Fig. 13.5d).

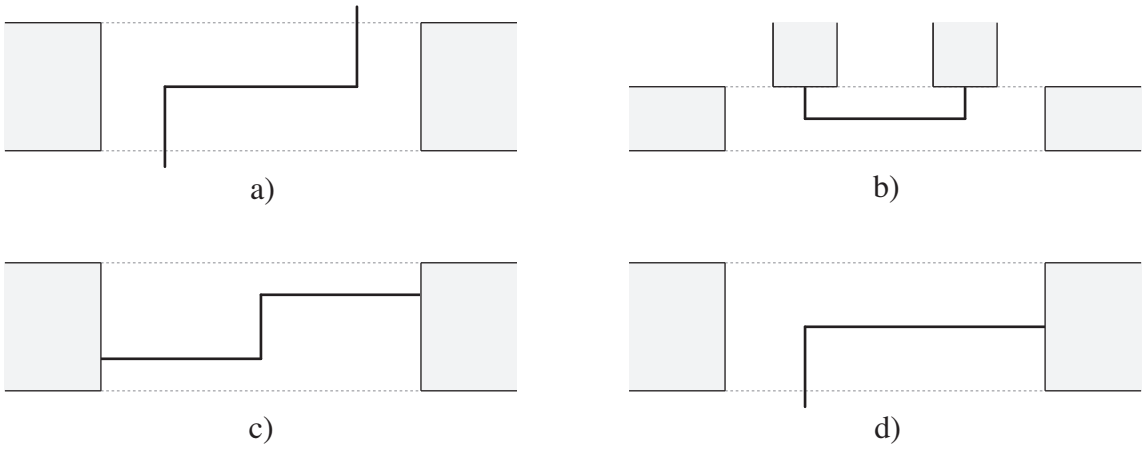


Figure 13.5: Possible constellation for the orthogonal routing

The path finding algorithm of Section 13.2 returns an ordered list of space tiles so that the predecessor or successor of a specific tile can be obtained from this list to identify the direction a line enters or leaves a tile. The actual direction can then be calculated by either using the neighbor finding algorithm to check which side of the current tile touches the next or previous tile or comparing the coordinates of the tiles.

The orthogonal routing algorithm is illustrated by means of Fig. 13.6 where points s and t should be connected by a rectilinear polyline. The path finding algorithm returned the list

$\langle T_1, T_2, T_3, T_4, T_5 \rangle$ marked by the gray shaded tiles in Fig. 13.6a). The basic idea of the orthogonal routing algorithm is to adjust the horizontal interval (or "window" [Miriyala et al., 1993]) through which a vertical segment has to pass while iterating over the space tiles. The number of bends in the line is minimized by routing the line without changing its direction as long as possible. We assume that the start point s is fixed so that the initial horizontal interval is reduced to the horizontal position of the start point (i.e., left boundary l and right boundary r of the interval are set to the x coordinate of s) as shown by the dotted black line in Fig. 13.6a). The next tile in the path is T_2 so that the horizontal segment through which T_2 can be reached from T_1 has to be determined.

This interval, represented by the dotted black lines in Fig. 13.6b), does not overlap with the previous interval of Fig. 13.6a) so that two new bend points have to be added to the line. Both new bend point are vertically centered in T_1 and the second one is horizontally centered within the horizontal interval. The horizontal interval remains the same for the next tile T_3 in Fig. 13.6c) because its width is only reduced but never increased (i.e., the left boundary l of the interval is only moved to the right and its right boundary r only to the left). Its width has to be reduced for the next tile T_4 by moving its right boundary to the left. The last bend point in the line is moved so that it is still horizontally centered within the interval. This is can be done without problems because the new and the previous interval do overlap. The next tile T_5 does not demand any action as the horizontal interval lies completely within the horizontal bounds of the tile. Finally, the interval is reduced to the horizontal position of the end point t because the position of this point is fixed (as part of the secondary notation). The intervals of Fig. 13.6e) and Fig. 13.6f) do not overlap so that two additional bend points are required.

Because of the organization of the corner stitching structure as maximal horizontal strips, it suffices to calculate the horizontal interval. A vertical interval is not needed because it is always defined by the height of the tile. Fig. 13.6 shows that the vertical crossing of a line through a tile is by far the most common situation (in fact the line crosses all tiles in Fig. 13.6 vertically) because the other possibilities of how a line can pass through a tile (i.e., situations b), c) and d) in Fig. 13.5) are all special cases. Only the vertical crossing requires the calculation and adjustment of the interval because the information that is needed to insert a new bend point can not be influenced by the previous or next tile in the other situations.

One drawback of the organization of the corner stitching structure as maximal horizontal strips is that it can, together with the concept of inserting additional bend points as late as possible, result in suboptimal lines. The problem occurs if the line passes through multiple tiles vertically and changes its direction in the last of these tiles. It is then only vertically centered within the last tile even though it should be vertically centered within the area that is defined by multiple tiles (in order to better exploit the available space and maximize the distance to the node it passes around). Fortunately, the line can easily be adjusted in a second step because the whole empty space is explicitly represented in the corner stitching structure.

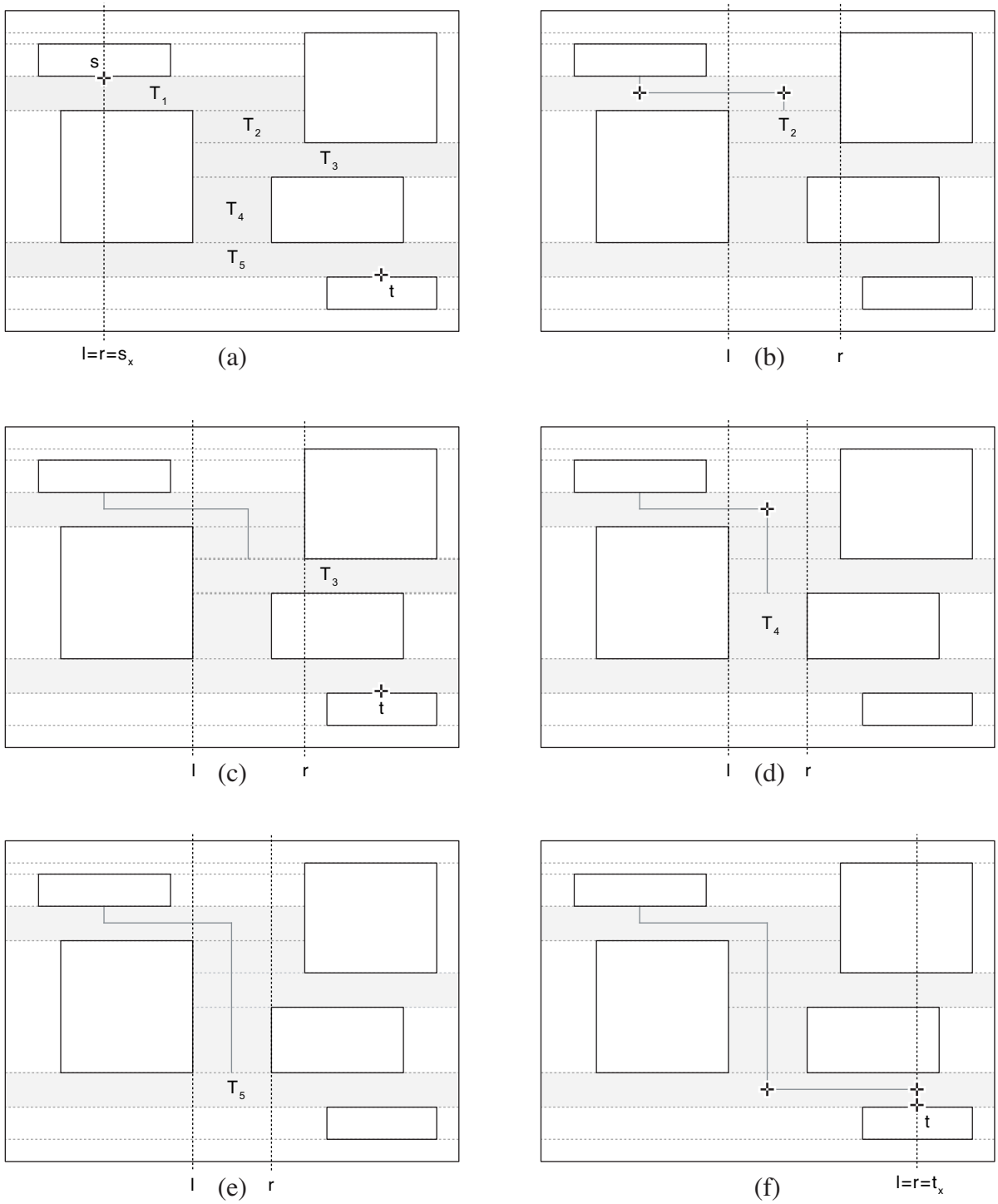


Figure 13.6: Calculate the bending points for an orthogonal line

Choosing the Best Solution

The path finding algorithm can, as already described in Section 13.2, deliver multiple possible solutions which are all optimal with respect to their length. The decision of which of these solutions should be chosen cannot be taken before the actual routing through these paths of space tiles has been done because the resulting line is not known before. Fig. 13.7 shows a situation for which the path finding algorithm returns two possible paths, represented by the gray shaded tiles in a) and b), from which one has to be chosen.

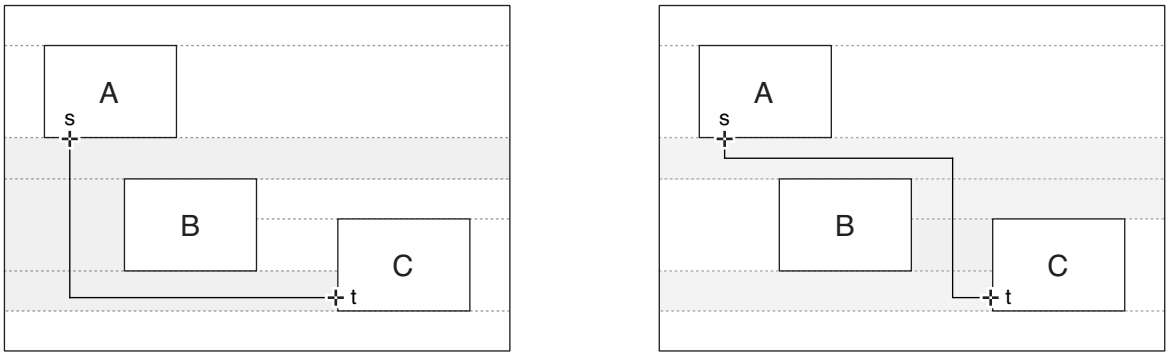


Figure 13.7: Choose among multiple possible solutions

The decision about which is the better solution can either be left to the user or done automatically by the algorithm. Letting the user decide increases on the one hand the influence the user has on the algorithm but on the other hand also increases the manual effort. A set of criteria has to be provided if the selection of the best solution should be done automatically by the routing algorithm. The number of bend points in the line (which favors solution a) over solution b) in the example of Fig. 13.7) or the uniformity of the segments' length can be used as criteria. The selection of one solution from a set of possible solutions is done completely independently of the optimization for the shortest path that is done by the path finding algorithm. Thus, it does not have any influence on this first implicit criteria of looking for the shortest path.

13.4 Line Crossings

As long as the lines are routed independently from each other, it is impossible to reduce line crossings or avoid overlapping line segments. But line crossings and overlaps reduce the readability of a diagram dramatically, because it becomes hard to follow the lines and to find out which nodes are connected. To maintain a good readability of the diagram, it is unavoidable to take existing lines into account while routing a new line (cf. Section 11.3.5).

It is possible to route the lines independently and reduce crossings afterwards by explicitly checking for intersecting lines and move the lines or line segments to reduce crossings or avoid overlaps. However, this solution can only provide local optimizations because the overall path is already defined. It is not possible to avoid areas that are already occupied by a lot of lines by looking for a detour through areas that are less crowded.

By extending the corner stitching structure and the path finding algorithm of Section 13.2, we can avoid line crossings during the routing of a line. We extend the corner stitching structure by a third tile type, the *line tile*. Line tiles are space tiles (or conversely node tiles that can be traversed by lines) weighted by a constant *cost factor* α. The algorithm now calculates cost values instead of distance values for the tiles. A higher cost factor α of a tile increases the costs for a line to pass through this tile. Thus, we transform the source distance σ and the distance δ into a *source cost* ω and a *cost* γ. Equations (13.3) and (13.4) show the extension of equations (13.1) and (13.2) with the cost factor α:

$$\omega(T_{next}) = \omega(T) + \alpha * \lambda(P, P_{next}) \tag{13.3}$$

$$\gamma(T_{next}) = \omega(T_{next}) + \lambda(P_{next}, t) \tag{13.4}$$

Fig. 13.8a) shows a snapshot after the second iteration through step 3 of the path finding algorithm of Section 13.2 extended by the cost factor, while routing a line from point s to t in order to connect nodes B and C. The cost $\gamma(T_2) = 311$ has already been calculated during the first iteration.

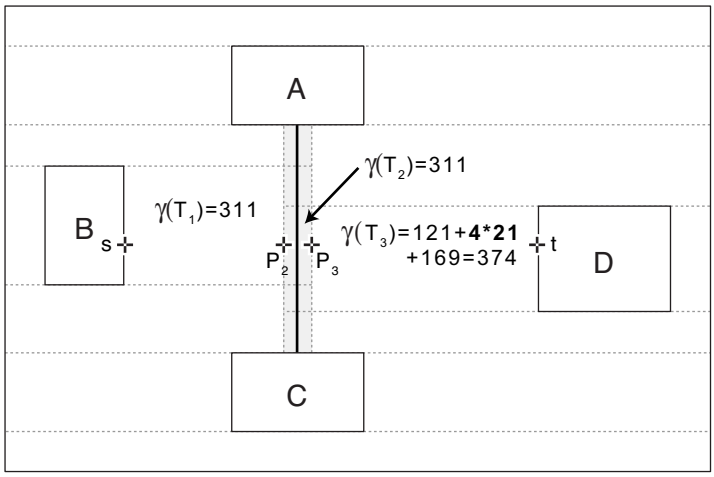


Figure 13.8: Weighted tiles

The costs for traversing tile T_2 increase by a cost factor α of 4.0 because tile T_2 is a line tile. Therefore, the Manhattan distance $\lambda(P_2, P_3)$ that is used to calculate the source cost ω for tile

T_3 is multiplied by the cost factor of line tile T_2. Adding the source costs ω to the Manhattan distance to the target results according to Equation (13.4) in a cost value γ of 374 for tile T_3. The cost factor α can now be used to express the detour of a line the user is willing to accept to avoid a crossing with another line. Increasing α in Fig. 13.8a) suddenly results in a line that is routed along the top boundary of node A because this path becomes "cheaper" than the path that crosses the existing line. The idea of tiles weighted by a cost factor is not restricted to line tiles. It may more generally be used to influence the routing of a line by indicating preferred areas or parts of the diagram that should be avoided.[2]

Step 4 of the path finding algorithm (cf. Section 13.2) has to be extended so that neighbors with the same *or a lower* cost value are selected during the retracing. The cost value becomes lower, if a line is crossed during the retrace. The drawback of this approach is that the number of paths that are found during the retracing increases because (i) the number of tiles increases with the additional line tiles and (ii) a path can go through any of these line tiles which results in a larger number of potential paths. To reduce the number of potential paths, we store the origins of a tile (i.e., the neighbored tile(s) from which the cost value of the actual tile has been calculated) during the expansion phase. We can then just follow these references during the retrace. A tile is added as origin to one of its neighbored tiles if the cost value that is calculated for this neighbor is equal to the current cost of the neighbor. If the calculated cost value is lower than the current cost value of the neighbored tile, all existing origins are removed.

This collecting of the origins during the expansion phase demands an additional invariant on the corner stitching structure in order to make the algorithm work in all situations as expected: A line tile can never have multiple tiles as left or right neighbors. If a line tile touches two or more tiles on the left or right side, it has to be split accordingly. This is the reason why there are five line tiles in Fig. 13.8a) instead of only one that covers the whole existing line. This invariant makes sure that the space tile that lie on the other side of a line are "visible" through the line so that each tile on the other side of a line can be reached directly without traveling a long way inside the costly line tile. Additionally, the orthogonal routing algorithm of Section 13.3.1 has to be generalized to work correctly on this extended corner stitching structure because (line) tiles are now frequently crossed horizontally (i.e., the situation of Fig. 13.5c) does now also occur when a line crosses an existing line). In addition to the horizontal interval, the algorithm has to maintain a vertical interval in order to route a line correctly in horizontal direction.

13.5 Calculation of the Start and End Point

So far we have assumed that the start and end point of a line are at fixed positions that lie on the boundaries of the nodes (or more precisely in the space tiles that touch the boundary of the node). That is the case for tools that offer a set of "connectors" or "magnets" on the boundary of a node that can be used as the start and end point of a line. This concept leaves some control over

[2] We use a cost factor α for each tile in our current implementation which is set to 1.0 for ordinary space tiles.

how the line should look to the user while it constrains the possibilities by offering only a small number of connectors. The decision on the position of the start and end point has to deal with a trade off between the amount of control left to the user and the part done automatically by the routing algorithm. On the one hand, the effort that is required to route a line should be reduced as much as possible which demands for a maximal degree of automation. The required effort can easily become unbearable in the case of a dynamic layout where the position and size of the nodes change while existing lines disappear and new lines emerge (cf. Section 11.2). But on the other hand, the position of the start and/or end point can be an important part of the secondary notation that is only known to the user. For example, the direction of the information seems to have a big influence on the understandability of a diagram (cf. Section 11.3.5) so that it can become important that a line ends at the top boundary of a node even though the shortest path ends at one of the other boundaries.

The concept of weighted tiles that was introduced to reduce the number of line crossings can also be used to calculate the start and end point of a line by projecting any point inside the node that has been selected by the user to the node's boundary. The basic idea is to start the line inside the start node (and no longer inside a neighbored space tile) and add a high cost factor to this start node so that the line tries to leave the node as fast as possible. Thus, the user can influence the location a line starts and ends at by setting this *click point* close to one of the node's boundaries. The routing algorithm then tries to find a compromise between starting the line as close as possible to the user's click point and the overall length and shape of the line.

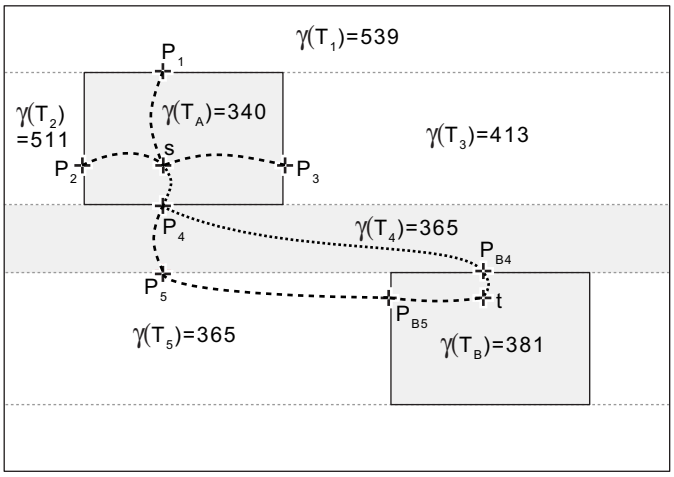


Figure 13.9: Calculating the position of the start and end point

Fig. 13.9 shows an example where the points s and t which lie now inside the start and end node have to be connected. The expansion phase of the path finding algorithm remains the same, but it now starts inside the start node and not in a neighbored space tile anymore. The costs $\gamma(T_A)$ are set to the Manhattan distance between s and t because the source distance $\omega(T_A)$ is initially 0. The first iteration calculates now the costs for the neighbors of tile T_A. The Manhattan distance from s to points P_1, P_2, P_3 and P_4 is, according to Equation (13.3), multiplied by a cost factor of

$\alpha = 1.8$. The resulting source costs ω are then added to the distance to the target which results in the costs $\gamma(T_1) = 539, \gamma(T_2) = 511, \gamma(T_3) = 413$ and $\gamma(T_4) = 365$ (the cost value is truncated to an integer during the calculation). Tile T_4 has the lowest cost value among A's neighbors because it can be reached by traveling the shortest distance inside the costly node tile T_A. As a consequence, the user can influence the selection of the space tile in which the line starts by selecting an appropriate click point inside the node. The path finding algorithm proceeds with tile T_4 as this is now the tile with the lowest cost value. In the next step, the end tile T_B can be reached from T_4 so that a solution has been found.

The decision on whether the current tile is now added as origin to the end tile is a little bit more complicated. The algorithm has to ensure that the position of the end point inside the end node has also an influence on the final line. The cost value for the end tile T_B is calculated by adding the Manhattan distance between P_4 and the point P_{B4} (i.e., the point inside T_4 that is closest to the point t) to the Manhattan distance between this point P_{B4} and t multiplied by the cost factor of T_B. We can ignore the distance to the target which is 0 because the target has been reached so that the cost for T_B is calculated according to $\gamma(T_B) = \omega(T_4) + \lambda(P_4, P_{B4}) + \alpha * \lambda(P_{B4}, t)$ or with the concrete numbers of Fig. 13.9, $\gamma(T_B) = 56 + 288 + 1.8 * 21 = 381$ (again truncated to an integer). The same calculation is done for the path that reaches T_B through T_5 which results in $\gamma(T_B) = 105 + 189 + 1.8 * 71 = 421$. This cost value is bigger than the one that was calculated from T_4 so that T_5 is not added as an origin to T_B. Therefore, the final path is P_A, P_4, P_B. A line through this path contains more bend points than a path that would go though P_A, P_4, P_5 and P_B (only one additional bend point instead of two) but the user has influenced the routing algorithm by his selection of the position of the start and end point. The fact that we store the origins of a tile during the expansion phase instead of just following the cost values during the retracing makes sure that a line cannot take any shortcut by jumping too early into the start node during the retracing and does indeed leave the start node through the boundary that was indicated by the user. The user can decide on how close to one of these boundaries a click point has to be in order to let this criteria win over the others (such as the number of bend points in the line) by varying the cost factor α of the node. Setting α to 1.0 reduces the user's influence as the user can no longer freely choose where the line should start and end but results in an optimal line with respect to the line's length and the number of bends.

We use a hybrid approach for the calculation of the start and end point in our current tool implementation. A high cost factor α is only used for the start or end node if the start or end point lies inside a border of predefined size around the inside of the node. We use a cost factor of $\alpha = 1.0$ (i.e., the same cost factor as for space tiles) for the start and end nodes of lines whose start and end points lie inside the node but not within this border. Thus, the click point of the user is only taken into account if the user places the start or end point close to the boundary of a node. This approach offers experienced users the possibility to make the position of the start and end point part of the secondary notation, while providing a pleasant routing for users that just click somewhere inside the node to define the start and end point (cf. Section 13.7 for a discussion of the secondary notation).

13.6 Reflexive Lines

The concept to start the routing of a line inside the source node and end it in the target node can also be used to routed reflexive lines or self loops (i.e., links for which the source node is also the target node). The basic idea is to split the node tile into three tiles: one that contains the start point, another one that contains the end point and an expensive tile in between. Fig. 13.10 shows an example with a reflexive line that starts and ends at node A. The node tile T_A is split into the three node tiles T_{A1}, T_{A2} and T_{A3}. T_{A1} contains the start point s of the line and T_{A2} its end point t. Tile T_{A2} is only one unit wide and has such a hight cost factor α that the line never passes through it. The unmodified line routing algorithm can then be used on this extended structure to route the reflexive line.

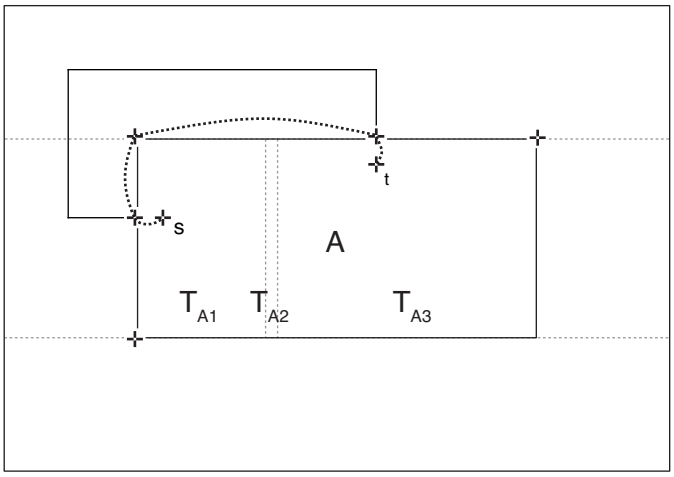


Figure 13.10: Routing of a reflexive line

The node tile can be split either horizontally or vertically. In our current implementation, we calculate the horizontal and vertical distance between the start and end point and then split along the direction with the longer distance. For example, the node tile of A in Fig. 13.10 is split horizontally because the horizontal distance between s and t is bigger than the vertical distance between these two points.

13.7 Discussion of the Routing Algorithm

The properties a line routing algorithm should have so that it can be used for dynamic hierarchical models were presented in Section 11.3. Each of these properties is now discussed for our routing algorithm.

Bypassing Nodes

Routing lines around nodes instead of crossing them is the rationale for a routing algorithm. Nodes and the white space between and around them are represented by different kinds of tiles in the corner stitching structure. The leeway for routing a line is then restricted to the white space tiles so that a line can never cross a node tile.

Short Lines

While the avoidance of nodes poses a constraint that cannot be violated on the routing algorithm, a minimal line length is the main optimization goal. Lines are kept as short as possible by applying a cost function and Dijkstra's shortest path algorithm to a non-uniform grid structure as shown in Section 13.4. However, the length of a line is not the only optimization factor that is taken into account while routing a line. The number of crossings with existing lines and the position of the start and end point can also influence the result as they are part of the cost function. Additionally, the solution found while minimizing the length of a line must not be unique (i.e., there can be multiple optimal solutions with the same minimal length).

Minimize Link Crossings

Our routing algorithm uses a cost factor to reduce the number of line crossings (cf. Section 13.4). Using a cost function instead of just avoiding line crossings completely makes it possible to look for a compromise between the length of the line and the number of link crossings. Both of these two characteristics of a line should be minimized. But minimizing the number of link crossings can result in a longer line and vice versa. By using a cost function, one can define the maximal length of the detour that is still acceptable to avoid crossings with existing lines.

Different Routing Styles

We have implemented only an orthogonal line router at the moment. Other routing styles can be implemented on top of our channel finding algorithm. Using routers that employ different routing styles was the main motivation to separate the channel finding from the actual routing (i.e., bend point calculation). However, some of the concepts such as the reduction of line crossings (cf. Section 13.4) are currently restricted to orthogonal lines (as the corner stitching structure is based on rectangles) and have to be extended to support straight-line or spline-based routings. For example, the approach of Marple et al. [1990] that uses trapezoids instead of rectangles for the corner stitching structure could be used.

Secondary Notation

In principle, our zoom algorithm lets the user influence the routing of a line by (i) selecting where the start and end points are anchored on the source and target node, (ii) varying the cost factor α to define to what extent the algorithm should avoid line intersections, (iii) manually selecting one solution from a set of possible solutions (or providing a selection function) and (iv) defining a point or a tile through which the line should pass. We have implemented the selection of the start and end point in the current version of our tool because this lets the user significantly influence the final shape of a line without too much effort (see also the discussion about the mental map of a line in Section 11.3.5). However, the experience with our tool has shown that the users do often not understand why lines do not always take the shortest route because they are not used to the click point concept. What they usually really want is more automation even though this results in a loss of control.

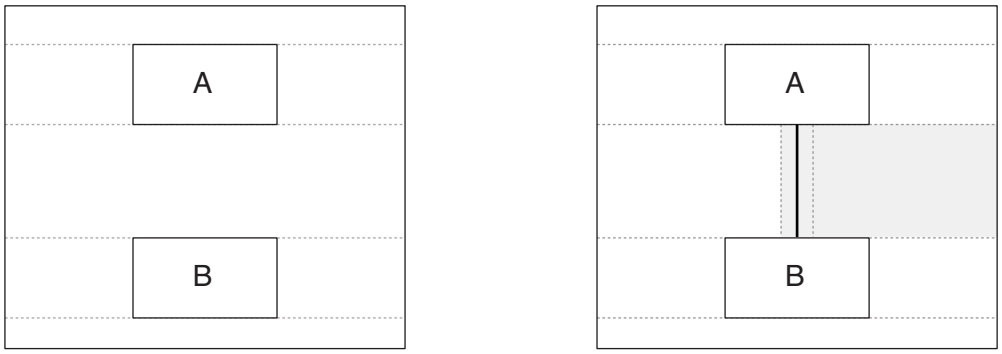
Preservation of the Mental Map

Our algorithm tries to preserve the mental map of a line in so far as it keeps the position of the start and end point which can be defined by the user invariant as long as possible. However, a line can (and indeed has to) look quite differently when the layout changes even though the position of the start and end point is fixed. A line can often be routed in a much simpler way (i.e., more direct with less bends) after a layout operation if the preservation of the start and end point is relaxed. For example, the line between nodes A and C in Fig. 11.3 on page 123 can be routed without an additional bend after the zooming of node B if the end point on node C can be changed. Keeping the position of the start and end point fixed at any cost often requires even more bend points in the adjusted than in the original layout. Thus the routing algorithm has to take other factors such as the length of a line into account while trying to preserve the mental map of a line.

Algorithmic Complexity

The routing algorithm must route the lines in real time whenever a modeler changes the layout by navigating, creating a view or editing. Since the wave expansion and retracing of a potential path takes place on the corner stitching structure, the number of tiles in this structure determines the complexity and thus the runtime of the algorithm. The number of space tiles in a basic corner stitching structure has the upper bound of $3n + 1$ for a layout with n nodes (cf. Section 13.1). Thus the algorithm has a linear runtime and space complexity with respect to the number of nodes. The calculation of the upper bound of nodes in a corner stitching structure which additionally contains line tiles is a little bit more complicated. Additionally to the space tiles, the line tiles have also to be taken into account because lines can pass trough them as well. Each (straight) horizontal or vertical line segment adds at most two additional tiles to the structure. This is illustrated in Fig. 13.11. The new line between nodes A and B adds two additional tiles

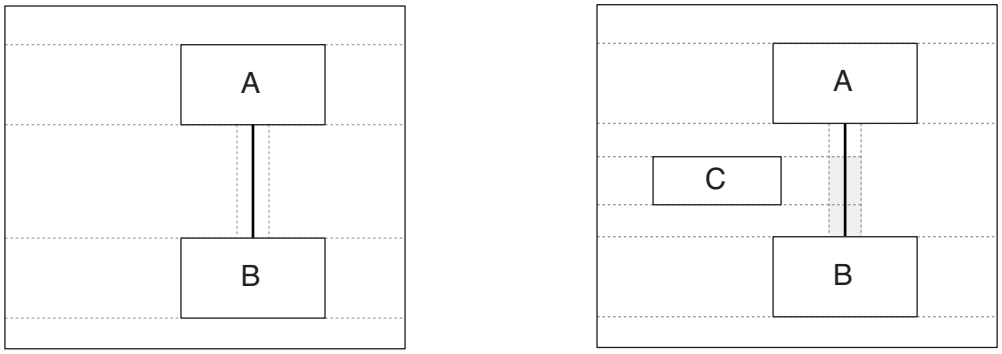
to the original layout of Fig. 13.11a): the line tile for the new line as well as an additional space tile that results from the splitting of the original space tile the line passes trough. The two new tiles are represented by the gray shaded rectangles in Fig. 13.11b).



(a) (b)

Figure 13.11: Additional tiles for a line segment

As soon as the diagram contains more nodes than just the start and end node of the diagram, these sibling nodes may result in additional line tiles. This is due to the constraint that a line tile can never have multiple tiles as left or right neighbors (cf. Section 13.4). Fig. 13.12 illustrates such a situation:



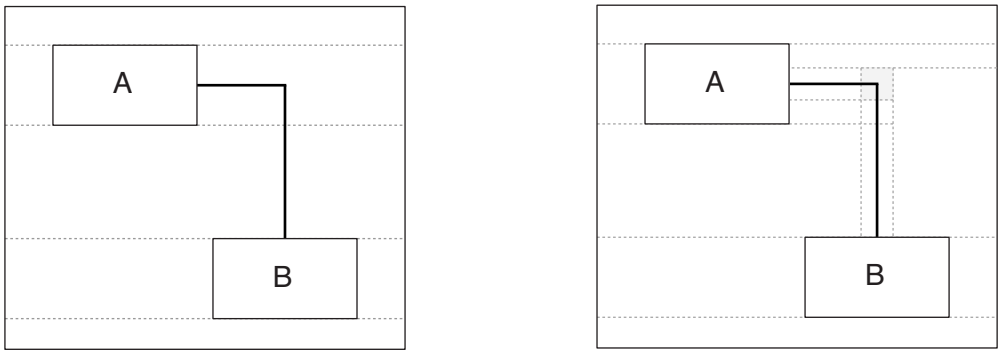
(a) (b)

Figure 13.12: Additional line tiles for a node

The insertion of node C into the initial layout of Fig. 13.12a) results in two additional line tiles in Fig. 13.12b). One of these tiles is the actual projection of node C while the other results from the splitting of the existing line tile. If the inserted node has a line to the left and to the right side, its insertion results in four additional line tiles. Due to the organization of the corner stitching structure in maximal horizontal strips (cf. Section 13.1) these additional line tiles can only occur

in vertical line segments. However, as a simplification so that we do not have to distinguish between vertical and horizontal line segments and because we are looking for an upper bound, we can state that each node can result in four additional line tiles.

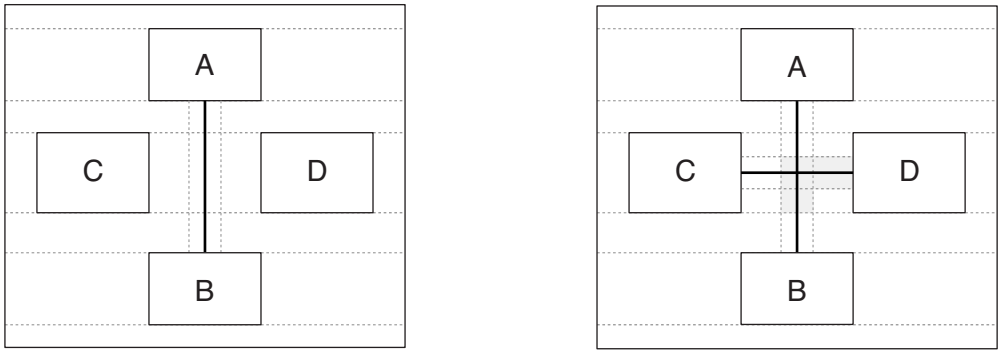
Fig. 13.13 shows that each bend (i.e., transition from a horizontal to a vertical segment or vice versa) results in one additional line tile (the gray shaded rectangle). The other line tiles result from the two line segments and from node A that splits the line tile representing the vertical line segment. Since a line tile has a minimal width[3] the tile representing the bend is never split.



(a) (b)

Figure 13.13: Additional line tiles for a bend

As soon as lines intersect, each such crossing results in at most three additional line tiles. As Fig. 13.14 illustrates, one of these additional tiles represent the intersection point itself, while the other two result from the splitting of the two line tiles representing the crossing lines.



(a) (b)

Figure 13.14: Additional line tiles for line crossing

[3]The width of the line tiles in the figures is that large only for illustrative purposes. Line tiles are actually only one unit wide.

Taking all these situations into account yields the following formula for the calculation of the upper bound t of tiles that are relevant for the routing of a line in a line tile structure:

$$t = 7n + 3c + 2s + b \tag{13.5}$$

where n is the number of nodes, c the number of crossings between existing lines, s the number of (straight) line segments all existing lines consist of and b the number of all bend points in existing lines. While this gives the upper limit for the number of nodes a corner stitching structure can consist of, the number of tiles that are actually visited during the wave expansion and retracing phases of the routing algorithm is much lower. First of all, the hierarchical nesting of nodes results in a number of smaller corner stitching structures containing only a fraction of the nodes and links of the diagram instead of one big structure that contains all nodes and links. Each line uses its own corner stitching structure that contains only the nodes and links that are, because of their position in the hierarchy, relevant for it (cf. Sections 11.1 and 13.1). Furthermore, due to the directed search employed in the wave expansion phase of the routing algorithm most of the time only a fraction of the tiles have to be visited until the target is reached. A rudimentary analysis of the line routing in three case studies[4] has revealed that in the average only 50-60% of the "routable" tiles (i.e., space and line tiles) are actually visited during the wave expansion. Existing lines increase the number of tiles but at the same time line tiles act as barriers baring the wave from propagating and thus reducing the overall effort of the routing algorithm. The average length of the tile paths that result from the retracing phase is also very low in the case studies as most of them consisted of three or less tiles so that the bend point calculation can be done very quickly.

[4]The case studies were the mine drainage and elevator system used for the experimental validation of the fisheye zoom (cf. Section 16.1) plus one real industrial case study, a model of a license management system consisting of 125 nodes and 108 links.

CHAPTER 14

Line Labels

A problem that is closely related to the line routing is the placement of labels that accompany lines. While line labels are not of much importance in abstract graphs, they often contain a big part of the information represented in a graphical model. For example, transition descriptions in statecharts [Harel, 1987] or ADORA models (cf. Section 4.1) contain a large part of the information represented in a diagram. Because of this large amount of information, line labels can have a significant size. This goes somewhat against the nature of node-link diagrams in which links only connect entities (the nodes) and are usually not seen as entities themselves. Some notations such as the Specification and Description Language (SDL) [ITU-T, 2000] address this representation mismatch by using a set of nodes (e.g., Input, Output, Procedure) instead of a link accompanied by a big label to describe a state transition. In contrast, the Requirements State Machine Language (RSML) [Leveson et al., 1994] represents state transitions by a simple arrow between two states and describes them externally to the diagram in a mixed textual/tabular notation. While this approach avoids the label problem almost completely as there are no transition labels anymore, it also removes one of the biggest advantages of a graphical state-based notation, namely the possibility to see the whole specification with one glance. Another possibility to reduce the number of line labels and thus potential problems with them is to show and hide them on demand or to show most of the information in a tool tip which pops up on close inspection of a line only.

Since the characteristics of (big) line labels are similar to those of nodes, the same problems of overlaps occur. Labels can overlap with existing nodes and/or other labels. These overlaps impede the readability of the diagram. Additionally, an inappropriate position of a line label close to other links can make it hard to determine the line it belongs to. Thus, deciding on the

position of a line label entails not only finding enough white space to avoid overlaps but also selecting an appropriate position relative to the line. Additionally, the position of a label relative to the line is often part of the secondary notation (cf. Section 2.5). For example, the position of association labels in UML (cf. Section 2.2.2) relative to the start and end node determines the association's direction they describe.

The dynamic layout that results from the use of a fisheye zoom (cf. Chapter 7) or view generation algorithm (cf. Chapter 8) intensifies the problem of overlapping labels. Labels can suddenly overlap after a layout adjustment. Fig. 14.1 shows an example. The label of the line between nodes B and C does not overlap with any other element in the original layout of Fig. 14.1a). In the layout that results from zooming out node A the label overlaps with nodes B and C because the zoom algorithm has moved the two nodes closer together as shown in Fig. 14.1b).

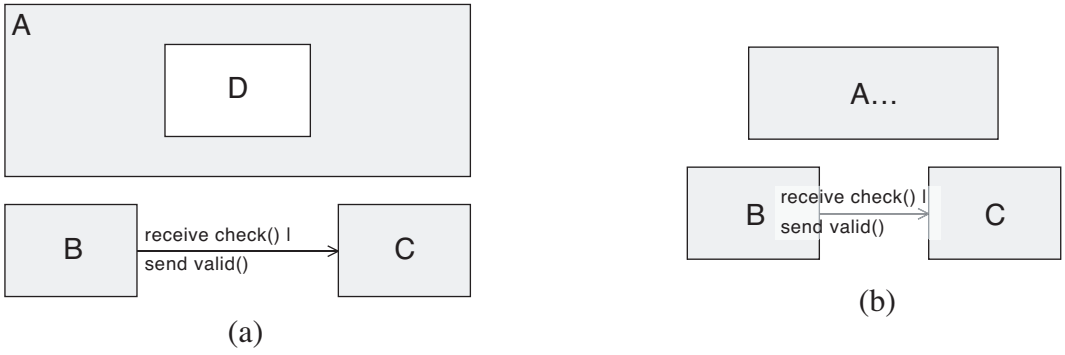


Figure 14.1: Overlapping label after zooming-out node A

14.1 Label Placement Rules

The problem of associating graphical features with text labels has applications in many fields such as graph drawing and cartography. Cartographers have been using rules for a good placement of labels over centuries. These rules can be used as a basis for an automatic label placement algorithm. For the problem of placing line labels in a diagram the following rules can be derived from the basic rules for placing labels on geographic maps [Imhof, 1975; Yoeli, 1972]:

1. Line labels must *not overlap* with other labels or other graphical features of the layout.
2. Each label can be easily *identified with exactly one line* (i.e., the assignment of a label to a line is unambiguous).
3. Each label must be placed in the *best possible position* (among all acceptable positions). The best possible position is usually defined by a set of aesthetic preferences such as that a label to the right is always preferred over one to the left of a symbol.

For diagrams that employ the nested set notation (cf. Section 3.3.1) an additional rule has to be added:

4. Each labels must be *fully contained* within the parent node of the line it belongs to. The parent node of a line is defined as the first common ancestor of the line's source and target node.

14.2 Basic Approaches

Algorithms to place[1] line labels can be classified according to the degree they influence or change the overall layout of a diagram. Approaches can range from the movement of nodes and thus significant changes of the layout to minimal invasive approaches which take the layout (i.e., position and size of the nodes) as well as the routing of the lines as given constraints.

14.2.1 Adjusting the Layout

The basic idea of a label placement algorithm that is integrated into a layout algorithm is to adjust the layout (i.e., position and size of the nodes) in order to get the space required by the line labels. One solution could be to use the zoom algorithm of chapter 7 to resolve overlaps between line labels and nodes or among line labels. This seems to be a straightforward approach since the zoom algorithm has been designed to avoid or resolve overlaps between nodes. Fig. 14.2 shows how this technique can be applied to the situation shown in Fig. 14.1b) to resolve the overlaps. Nodes B and C have been moved away from each other in order to provide the space required by the label.

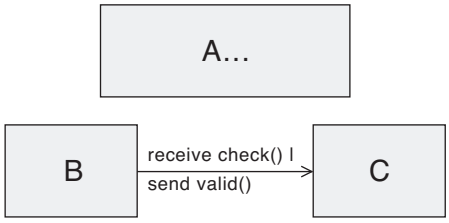


Figure 14.2: Move nodes A and B to provide the required space to the label

However, this approach has three drawbacks. First, the additional layout changes that occur during the label placement can easily destroy the secondary notation and the user's mental map of a diagram (cf. Sections 2.5 and 5.3). Second, it can become difficult to maintain the stability of

[1] We use the term label "placement" not only for the calculation of a label's position but for the definition of its boundaries (i.e., position *and* size) in general.

the zoom algorithm (cf. Section 5.4) if line labels have an influence on the layout. For example, if the user zooms a node in and then out again, the resulting layout changes have to be undone. This can become rather difficult if the layout is changed in between to provide additional space to line labels. And third, as shown in Section 11.2, the layout adjustments done by the zoom algorithm often demand for a different routing of a line even though the overall layout has not changed dramatically. To handle such situations, we have separated the line router from the zoom algorithm so that the line router adjusts the lines after the zoom algorithm has adjusted the layout. If the zoom algorithm is additionally responsible for adjusting the position of line labels, it tries to keep their relative position to the surrounding nodes in order to preserve the mental map. If a line is then routed again, the position of accompanying labels and the line often do not match anymore. Using the zoom algorithm to place labels has an influence on how the line is routed afterwards (because it changes the layout and thus the position and/or size of the nodes which determines the route of the line). This routing has again an influence on where the label should be placed (as the line and the label should be close together). Thus, coupling the zoom and routing algorithm by using the zoom algorithm to place line labels lets them influence one another which may result in an oscillating layout. An additional, more technical advantage of separating the routing and label placement algorithm from the zoom/layout algorithm is that they can be used and maintained independently.

14.2.2 Influence the Line Routing

A little less invasive is the idea to integrate the placement of labels into the line routing algorithm. Such an integration employs two different ideas: First, to integrate existing, already placed line labels (i.e., those belonging to lines which have already been routed) into the routing algorithm so that they are not crossed by newly added lines. This can be achieved by treating line labels in a similar way as existing lines in our routing algorithm (cf. Section 13.4). And second, to let the routing algorithm look for another solution if the current route of a line does not provide enough space for the associated labels. Fig. 14.3 illustrates how the line between nodes B and C can be routed to avoid the overlaps shown in Fig. 14.1b):

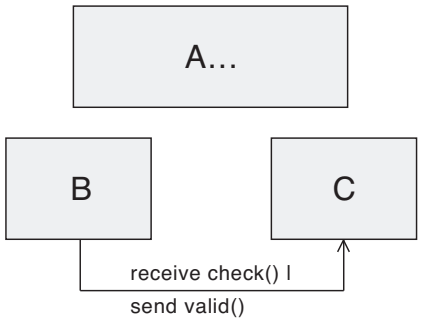


Figure 14.3: Reroute the line to provide the required space to the label

The biggest problem of this approach is that it can influence the line significantly and thus make it harder to fulfill some of the desired properties for the line routing, such as the minimal length of a link (cf. Section 11.3.2). On the other side, the interferences with the secondary notation and the mental map are much less severe than for the integration of the label placement into the zoom algorithm because lines are usually seen as less crucial than nodes for the preservation of the mental map (cf. Section 11.3.5).

14.2.3 Layout and Line Routing as Constraints

The most conservative label placement strategy is to place the labels without changing anything of the existing layout of nodes and lines. Overlaps between labels and nodes or among labels can be avoided by adjusting the labels' position and size only. Thus, the label placement algorithm works on a fixed geometry. While such an approach minimizes the influence on the user's mental map and secondary notation, it also constrains the freedom of the label placement algorithm. The chance that the algorithm cannot place a label without any overlaps is significantly higher than with the other approaches. However, this problem can be mitigated if the model is created interactively by the user as he can influence and thus improve the label placement by adjusting the layout himself.

The cartography field has to deal with similar label placement problems: While the "node label placement problem" deals with the labeling of points in the plane so that no labels overlap points or other labels, the "edge label placement problem" is concerned with the assignment of text labels to edges such that the quality of the assignment is optimal. Even though the label placement problem is NP-hard and essentially unsolved [Kakoulis and Tollis, 1997], a large number of heuristic approaches have been developed (see e.g. [Christensen et al., 1995] for a survey of algorithms for the labeling problem). While these algorithms use optimization techniques such as discrete gradient descent or simulated annealing to find a global optimum, such techniques cannot be used for an interactive modeling tool because the runtime of a label placement algorithm is a critical factor as a potentially very large number of labels have to be placed without a remarkable delay.

14.3 Tile-based Label Placement

The corner stitching structure which is the basic data structure of our routing algorithm (cf. Section 13.1) can be reused for a simple label placement algorithm. Because the general label placement problem is NP-hard, the presented approach is a heuristic which does not always result in an optimal solution. Furthermore, because a graphical model is usually created incrementally by the user adding one element after the other, the presented technique is also incremental. The labels that belong to a line are placed whenever this line is routed and the label placement algorithm works within a fixed geometry defined by the nodes, previously routed lines and the labels

which belong to previously routed lines. Thus, already placed labels are not repositioned in order to reach a global optimum like in other labeling algorithms such as [Christensen et al., 1995] or [Kakoulis and Tollis, 1997]. The presented algorithm has already been published in [Reinhard and Glinz, 2010].

The basic idea of the label placement approach is to use the corner stitching structure to calculate the white space that can be used to place line labels. This calculation is trivial because the empty space is explicitly represented in the corner stitching structure (cf. Section 13.1). The white space calculation avoids overlaps of line labels with nodes, existing lines (if they are explicitly represented in the corner stitching structure as shown in section 13.4) and existing line labels (if they are represented in the structure in similar way as existing lines). Additionally, the use of the corner stitching structure results in labels that are always contained within the line's parent node. A set of potential label positions with a predefined relative position to the line is then calculated within this white space. Among these candidate positions the best one with respect to some predefined criteria is selected. The detailed steps of the algorithm are the following:

1. Iterate over all (vertical and horizontal) line segments.

2. Use the corner stitching structure to calculate the available free space around the segment: First a reference point for each label is defined. This point can either be the midpoint of the segment if the label should be shown on the middle of the line or the segment's start or end point. The corner stitching structure is then used to find the tile this reference point lies in (cf. Section 13.1 for the detailed algorithm to find a tile at a given location). Finally, the free space is calculated by vertically extending the boundaries of the tile containing the reference point. Fig. 14.4 shows an example of this step. P_1 and P_2 are the reference points for the two line segments of the line connecting nodes A and B.

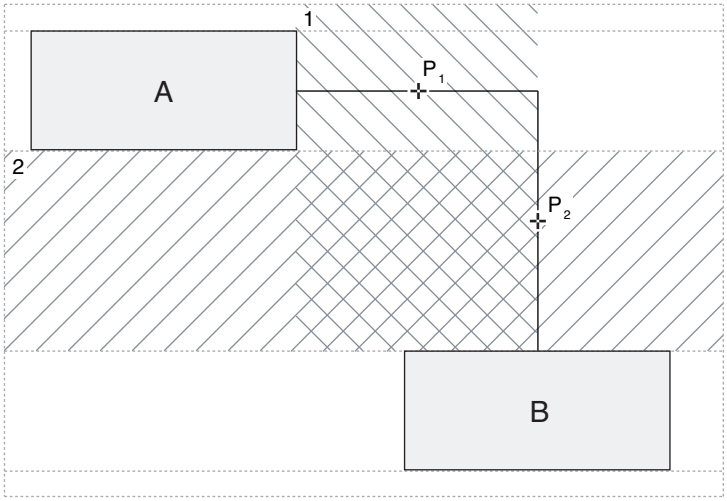


Figure 14.4: Calculation of the label space on the corner stitching structure

The corner stitching structure for this constellation is shown by the dashed lines. The hatched areas 1 and 2 show the free space that has been calculated for P_1 and P_2 respectively. In our current implementation the width of the free space for horizontal line segments is restricted to the length of the segment (as shown for the hatched area 1 in Fig. 14.4). The vertical expansion of the free space stops as soon as the width of the next tile above or below is smaller than the current width of the white space. That's why the expansion of the free space 2 stops below node A and above node B.

3. Determine three candidate positions for each line segment: For a horizontal segment these are above, below and vertically centered around the reference point. For a vertical segment they are to the left, the right and vertically centered around the reference point.

   Fig. 14.5 shows the six candidate positions for a line label that belongs to the line between nodes A and B. In our current implementation the label is asked for its required height given a specific maximal width. This maximal width is directly taken from the width of the free space adjusted by the relative position of the label (i.e., the maximal width of label 6 in Fig. 14.5 is the width of the free space to the right of the second line segment). This is a very basic approach which tries to make labels as wide as possible. Other more sophisticated approaches that take for example the content of the label into account to calculate its boundaries can easily be implemented as part of the label's internal size calculation without changing the placement algorithm. For example, in ADORA models such an approach could always try to write each of the components of a transition description on a separate line.

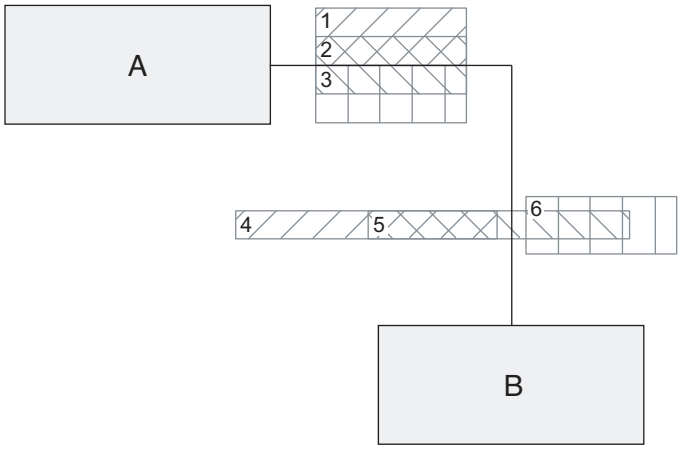


Figure 14.5: Candidate label positions

4. Select one of these candidates: To do so we associate a score with each candidate position. This score combines a set of a priori preferences with a value that reflects the severity of the violation of the basic label placement rules (1) and (2) of section 14.1. The basic label placement rules can be violated if there is not enough space for the label at the given

position so that it overlaps with nodes (or other labels) or if the label is crossed by the line it belongs to.

For the example of Fig. 14.5, the algorithm selects label position 1 if it prefers labels above horizontal segments or position 4 if it prefers labels to the left of vertical segments. The scoring can, together with a specific reference point, also be used to prefer positions close to the source or target node if such a position is an important part of the secondary notation.

Discussion

The implementation of the presented label placement algorithm is in its basic form straightforward as a large part of the logic lies in the corner stitching structure. As a proof-of-concept, it shows that it is possible to use the corner stitching structure, originally used for the line routing, for the placement of labels too. Much more sophisticated approaches can be built on top of this basic idea. The three main extension points are the calculation of the white space (e.g., calculating the free space for each candidate position independently), the calculation of the candidate positions (e.g., using multiple reference points or different label boundaries for each segment) and the scoring of the candidate positions. However, already the presented simple approach can mitigate the label problem to a large extend even though it may not always find the "best" position for a label. Fig. 14.6 shows how our current implementation solves the problem of overlapping labels after a zoom operation as shown in Fig. 14.1 by adjusting the boundaries of the label:

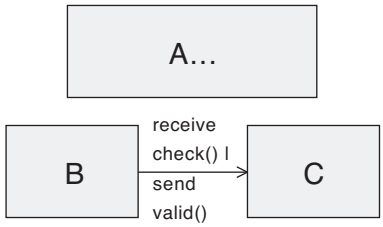


Figure 14.6: Solution for Fig. 14.1

Part IV

Validation

CHAPTER 15

Constructive Validation

The basic goal of the work presented in this thesis is to show that the visualization concepts that are an integral part of the ADORA modeling approach can be realized and made to work in a basic implementation. Thus, their implementation in a working tool and use on models of realistic size constitute an important part of the validation. While Chapters 10, 13.7 and 14.3 discuss technical aspects of the zoom, line routing and label placement algorithms and data structures in order to show that the presented concepts work and have some desired properties, this chapter gives a short overview over their actual implementation in the ADORA tool.

15.1 Layout Functionality

We use the model of a simple elevator control system to illustrate how the ADORA tool supports the modeler in editing and navigating through a model. This model is the same as the one used for our experimental validation (cf. Chapter 16). It is shown with all views enabled and all nodes zoomed in by Fig. A.2 on page 195. Even though the model is with its 67 nodes and 63 links still of reasonable size, it can easily be seen in Fig. A.2 that it already becomes impractical to show (and especially print) the whole diagram with all elements at once. As a consequence, the screenshots in the following sections show only a subset of the elements in the model. However, the underlying model is the complete one, so that the diagram looks exactly like the one in Fig. A.2 if the user zooms in all nodes and enables all views. The usefulness of the screenshots would be rather limited without the complexity management mechanism employed by the fisheye zooming and dynamic view generation implicitly being used all the time.

Fisheye Zooming

The fisheye zoom algorithm is implemented in the ADORA tool as far as described in Chapter 7. Model elements that support a hierarchical nesting (e.g., abstract objects and states) can be zoomed-in and -out by simply clicking on them. Thus, the user can freely navigate through the hierarchy of the model. Fig. 15.1 shows the basic and structural view of the elevator control system on a high level. Only 20% of all model elements are visible in this view as most of the nodes are zoomed out. The modeler can see all the components constituting the *Car* component together with the elements in its context they are related to without being overwhelmed by all the internal details of these components. Thus, the ADORA tool can help a lot in gaining a global or local (i.e., of a specific part of the model) overview over a big and complex diagram. The tool can adapt the layout (i.e., change the size and position of the nodes and reroute the lines) in real time without a noticeable delay. However, the layout adaption is animated and thus artificially slowed down in order to help preserving the mental map while adjusting the layout (cf. Section 5.8).

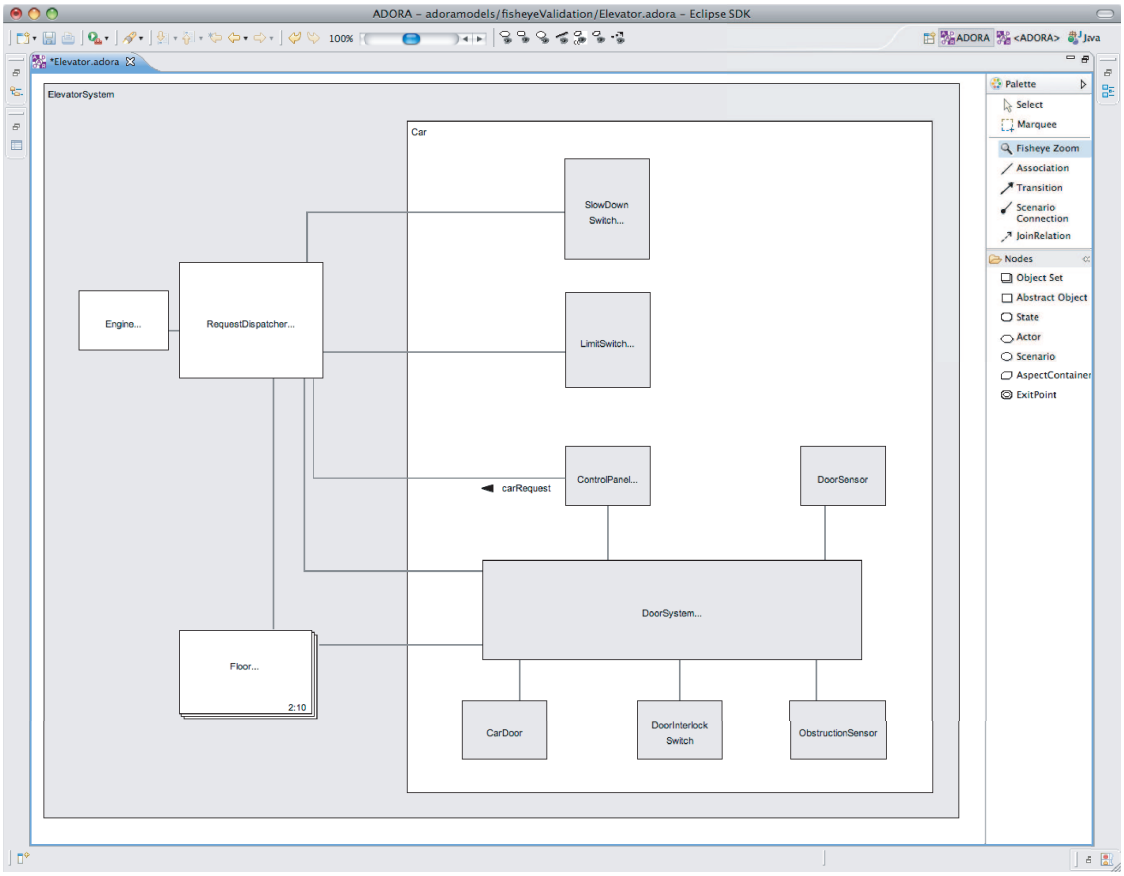


Figure 15.1: Base and structural view of the elevator system

15.1.1 Dynamically Generated Views

A separation of concerns is achieved by letting the user disable or enable any of the views of the ADORA language (except for the base view which is shown all the time). Views are enabled or disabled globally (e.g., all states and transitions become visible if the behavior view is enabled). However, the final appearance of a diagram results from combining the view concept with the fisheye zoom because model elements that are located within a zoomed-out node do not become visible even if the view they belong to is enabled. The ADORA tool relies on the concepts presented in Chapter 8 to adjust the layout if model elements are shown or hidden according to the selected view(s). Fig. 15.2 shows again the same model but this time with the focus on the behavior of the *UpButton* within the *Floor*'s *ControlPanel* component.

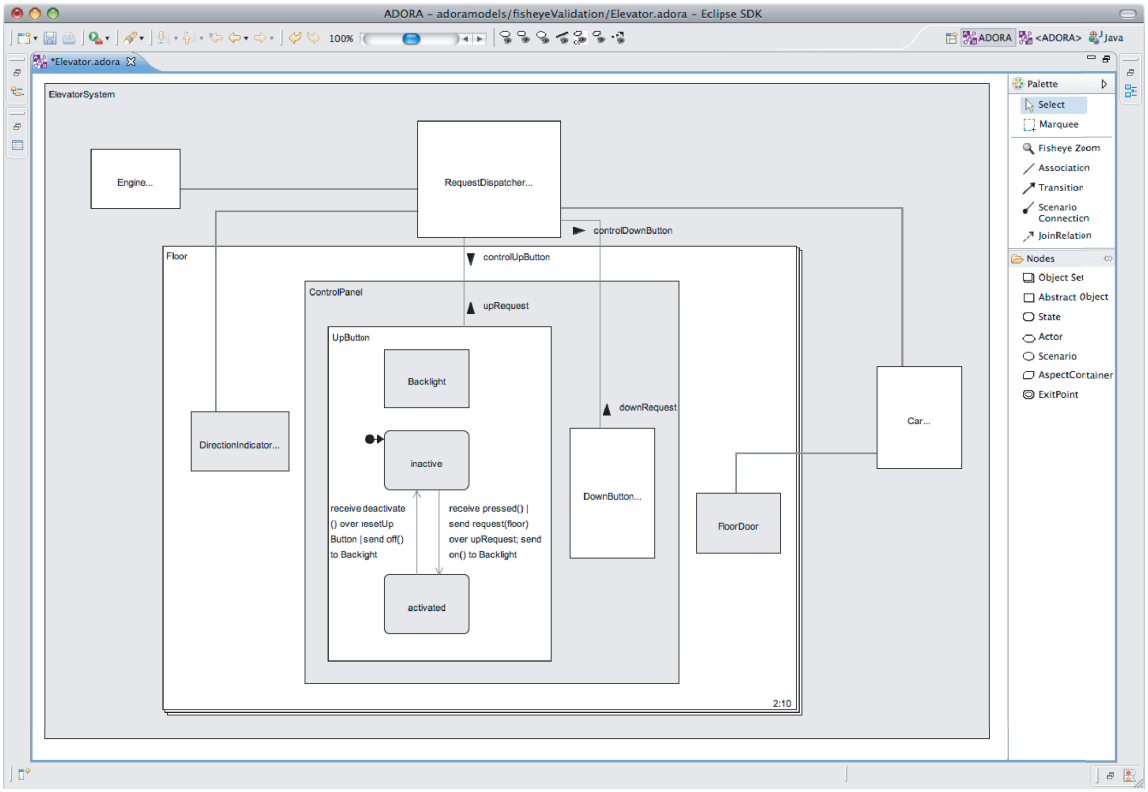


Figure 15.2: Behavior view with focus on the UpButton

To produce this view, the user has enabled the behavior view and zoomed out the neighboring nodes. The diagram in Fig. 15.2 is still compact and concise even though it is a very detailed view as it shows the formal transition description and is thus on the same level as the program code. In contrast to other modeling tools, the description of the behavior of the *UpButton* is shown in the context of this component. As a consequence the user is always aware of the context of the component she is currently working on (e.g., the *UpButton* in the *ControlPanel* of a *Floor*). Additionally, some of the elements in the context have a direct influence on the behavior of

the *UpButton* within the context of the elevator system. For example, the transition from the *activated* to the *inactive* state is triggered by an event which is sent over the communication channel between *RequestDispatcher* and *UpButton*. The user can now further investigate the collaboration between these two components by zooming into the *RequestDispatcher* to find out where exactly the triggering event is produced. The transition between different diagrams that result from the enabling of different views is animated to increase the continuity.

15.1.2 Editing Support

The nested node notation employed by the ADORA language to represent the hierarchical decomposition increases the need for tool support because a change in one part of the model is propagated upwards through the hierarchy to the root node. For example, the user has to expand all direct and indirect parents of a node to provide the space that is required by a new node. In order to make the nested node notation usable without any overhead, the ADORA tool automatically expands ancestor(s) if more space is required after a node has been inserted or moved by using the mechanism presented in Chapter 9. Conversely, parent node(s) are contracted automatically by the tool if a node is removed. This editing support of the the ADORA tool is illustrated in Fig. 15.3.

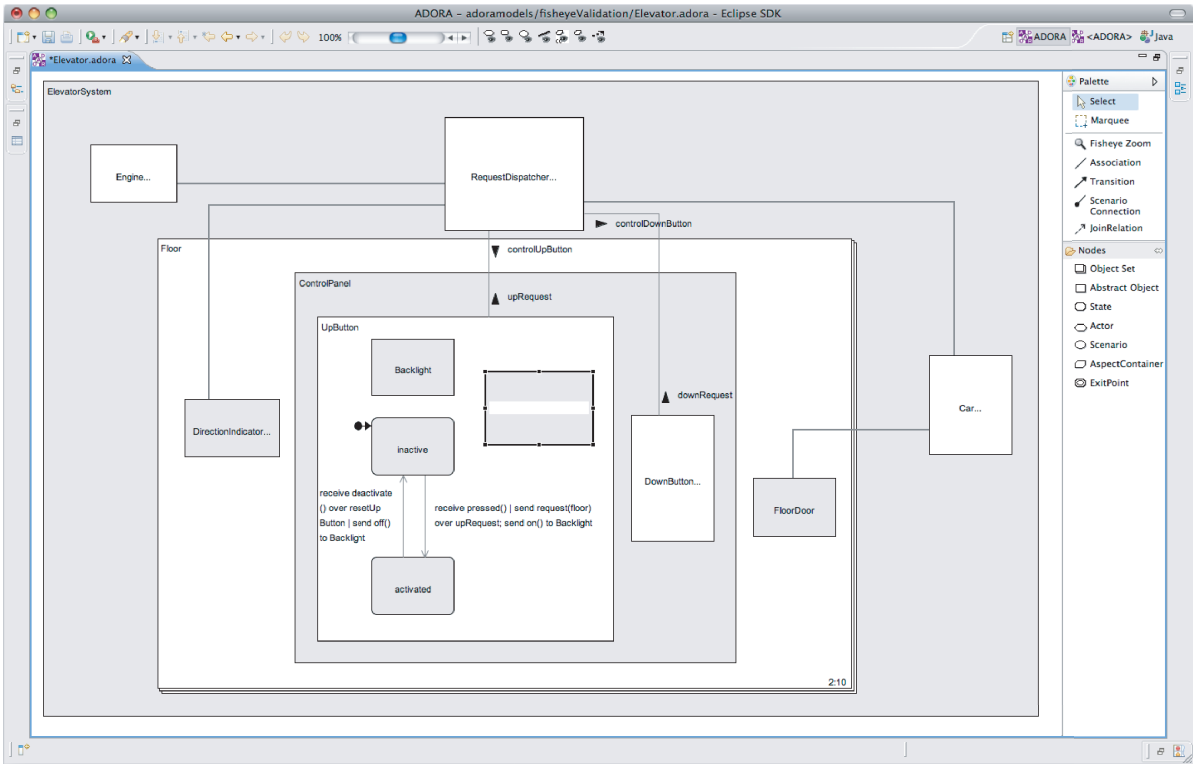


Figure 15.3: Inserting a new node

The insertion of a new component into the *UpButton* component of the initial layout of Fig. 15.2 expands all the ancestors of this node in order to provide the required space. This layout adjustment tries to preserve the original layout as far as possible so that the user still recognizes the diagram.

15.1.3 Line Routing and Label Placement

Lines are routed completely automatically by the tool (as described in Chapter 13) to unburden the user from this tedious task which has to be performed after each change in the layout. Such layout changes occur frequently because of the fisheye zooming and the dynamically generated views. Furthermore, abstract associations (cf. Section 11.1) are calculated automatically whenever at least one of the direct or indirect parents of a line's source or target node is zoomed-out. The user has some influence on how a line looks like by locating the start and end point of a line. Furthermore, labels attached to the lines are placed automatically (cf. Chapter 14). Fig. 15.4 shows the resulting layout after a link between *DirectionIndicator* and *FloorDoor* has been added to the layout of Fig. 15.3:

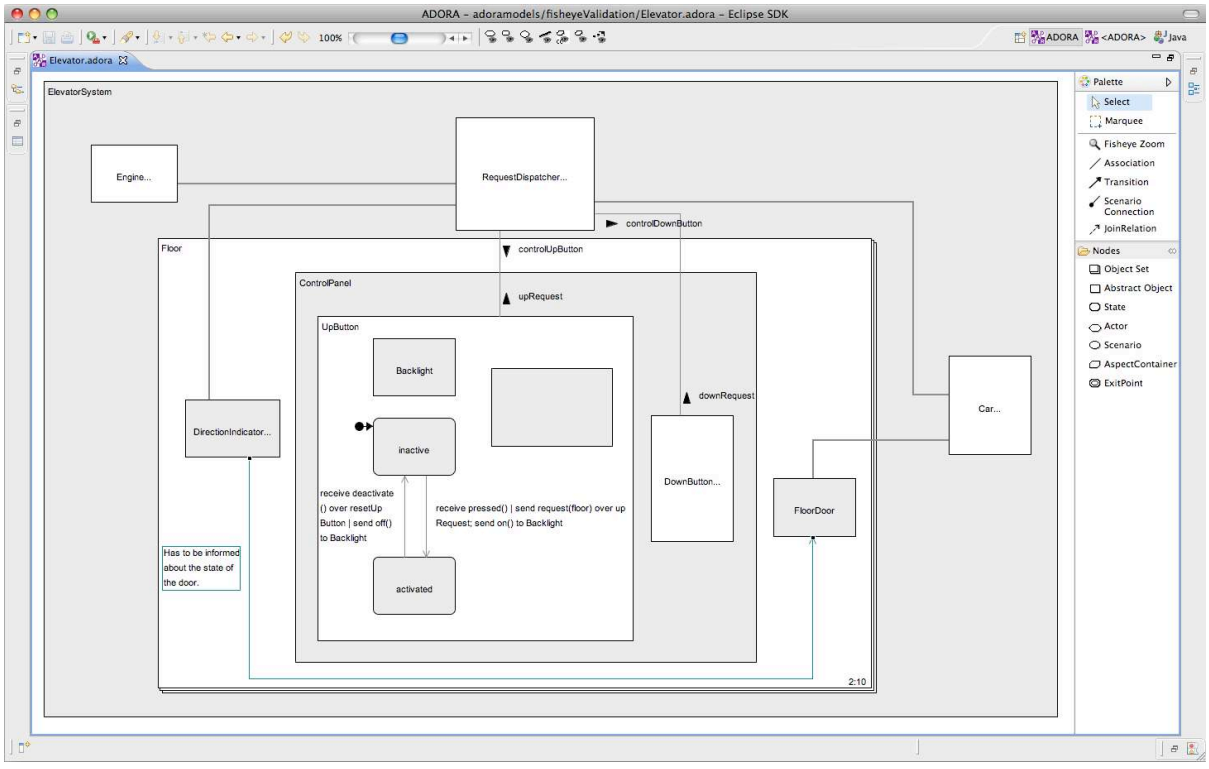


Figure 15.4: Automatic line routing and label placement

The link avoids the obstacle (i.e., the *ControlPanel* component) between its start and end node while staying within the boundaries of the common ancestor of the start and end node (i.e., the

Floor component). The alternative route around the top boundary of the *ControlPanel* has been discarded during the routing as it would have crossed two existing links. Fig. 15.4 also illustrates how the alternating shading of nodes (cf. Section 3.3.1) helps in distinguishing links from node boundaries. The label that accompanies the new link has been placed at the position providing the most empty space. Changing the horizontal position of the link's start point within the *DirectionIndicator* component (which has been defined by the user by clicking on the appropriate position within the node) can change the label's position when the empty space at its current position shrinks.

15.2 Technical Aspects

Even though we used the ADORA language and tool as an example to show how the concepts presented in this thesis can be used on an integrated modeling language, the concepts themselves are independent of any specific language. Their implementation in the ADORA tool is a proof of concept. The only language or more precisely representation concept that is required by the fisheye zoom is the nested node notation (cf. Section 3.3.1). The routing algorithm relies only on the basic concept of node link diagrams (cf. Section 2.1.3). These requirements of the zoom and routing algorithm on nodes and links are described in a few simple Java interfaces. The fisheye zoom and line routing components are decoupled from the core of the ADORA tool so that they can be used in other environments too. The code base of the two components is rather small: The zoom component has roughly 5200 TLOC (+ 4800 TLOC of unit tests) and the line routing component has 2900 TLOC (+ 3600 TLOC in tests).

CHAPTER 16

Experimental Validation

While the focus of the last chapter lies on the implementation of the presented concepts and algorithms in order to show that they do work on models of realistic size, this chapter examines the overall usefulness of the presented approach. We have conducted a small controlled experiment (i.e., a classroom experiment with students) to investigate the use of the fisheye zoom technique. The influence of our line routing technique (cf. Part III) was not addressed explicitly during the experiment. However, it was implicitly part of the experiment as all lines were routed automatically with the presented technique during the experiment.

16.1 Experiment

The aim of our experiment was twofold: First, to compare the performance of using a graphical model with a fisheye zoom with the one of using a fully expanded model to find out whether there are any significant differences. And second, to observe how exactly the subjects use the fisheye zoom (i.e., which zoom operations are executed in which order). We did not compare the fisheye zoom with an explosive zoom because (i) this comparison has already been done [Schaffer et al., 1993, 1996] and (ii) some constructs of our technique such as abstract relationships (cf. Section 11.1) cannot be used in combination with an explosive zoom. Additionally, a fully expanded model makes a stronger benchmark for a comparison since the mental integration of different models, which is the biggest drawback of an explosive zoom, does not apply. The major drawback of comparing our fisheye zoom to a fully expanded model instead of the traditional explosive zoom is that it almost certainly results in much less significant results than for example

the experiments of Schaffer et al. [1996].

The test subjects were asked a set of dual- and multiple-choice questions about shown models. They were split into two groups of which one group had to answer a set of questions about a model using the fisheye zoom while the other group had to answer the same questions about the same model without the possibility to use the fisheye zoom (i.e., without the possibility to open and collapse nodes). The roles were then exchanged so that the first group used a fully expanded model without the possibility to zoom while the second group had to use the fisheye zoom to answer the questions about a second model.

The two factors efficiency and effectiveness were used to evaluate the performance. The efficiency was determined by measuring the time used to answer a question. We define the effectiveness by whether, or more precisely to which degree, a given answer was correct. We combined the two factors in order to judge the performance by using the ratio between efficiency and effectiveness which gives us the number of correct answers per time unit[1]. We use the term performance points for the resulting value.

$$\text{Performance points} = \frac{\text{Correctness in \%}}{\text{Required time in seconds}}$$

Correctness lies somewhere between 0% and 100% for multiple-choice questions and is 0% or 100% for single-choice questions. The testing environment ensured that the user always selected at least one of the given answers so that the situation of not answering a question did not occur. In addition to these objective questions, the subjects were also asked a set of subjective questions.

Hypothesis

Our null hypothesis was that there exists no difference in performance (with $p = 0.05$) between subjects using our fisheye zoom and those using the fully expanded model. The performance of the subjects was measured by calculating the performance points as described above.

Case Studies

We used two independent case studies for the experiment. Both of these case studies must have similar size and complexity. A subset of the ADORA language (cf. Section 4.1) was used as notation for the models because we have implemented our zoom algorithm as part of the ADORA tool (cf. Section 4.2). Only the base, structural and behavior views of ADORA have been used in the experiment to reduce the impact of the language and the notation on the results. The two case studies were the mine drainage and the elevator system.

[1] We follow here the approach of [Meier, 2009].

The *mine drainage system* concerns the software necessary to manage a simplified pump control system for a mining environment. The pump is used to pump mine water, which collects at the bottom of a shaft, to the surface. The main safety requirement is that the pump must not operate when the level of methane gas in the mine reaches a critical value due to the risk of explosion. Additionally, the system must measure the level of carbon dioxide in the air and detect whether there is an adequate flow of air. An alarm must be activated if the carbon dioxide level or airflow become critical. The mine drainage system case study is widely used in the software engineering field and goes back to Kramer et al. [1983]. The model is shown in Fig. A.1 as part of Appendix A. It consists of 55 nodes (14 components and 41 states) and 63 links (13 associations and 50 transitions).

The second case study, the *elevator system* (which was originally introduced by Filman and Friedman [1984]), comprises the control software of a simple single car (cabin) elevator system. Parts of the system are the different operation units in the car and on the floor as well as the control of the door mechanism and the motor. Fig. A.2 in Appendix A shows the model of the elevator system that was used for the experiment. The model of the elevator system contains 67 nodes (33 components and 34 states) and 63 links (19 associations and 44 transitions).

Test Environment

The ADORA tool (cf. Section 4.2) that was used for the experiment facilitates inquiries with a built-in electronic questionnaire. The model(s) the user interacts with and the corresponding questions are shown together at the same time. The given answers and the time needed to answer a question are recorded in a database which can subsequently be used to analyze the results. The screenshot of Figure 16.1 shows a question in the left panel together with the model that has to be used to answer the question in the center.

Questionnaire

The complete questionnaires for the mine drainage and elevator system can be found in sections A.2.2 and A.2.3 respectively. The questionnaires incorporate three kinds of questions: First, the subjects where asked about their gender and for a self-assessment of their modeling skills. Second, they had to answer a set of objective questions about the shown models. About half of these questions concerned only the semantics of nested node-link diagrams (e.g., whether two nodes are connected or which nodes are contained within a specific node). The other half concerned the ADORA execution semantics (e.g., which events are fired or the states a component is in after a set of events have occurred). The required operations to answer the questions were typical task to explore a graphical model. They were restricted to navigating through the model without doing any modifications. At the end of the experiment, the subjects had to judge with their subjective opinion the usefulness of the fisheye zoom and an (optional) linear zoom.

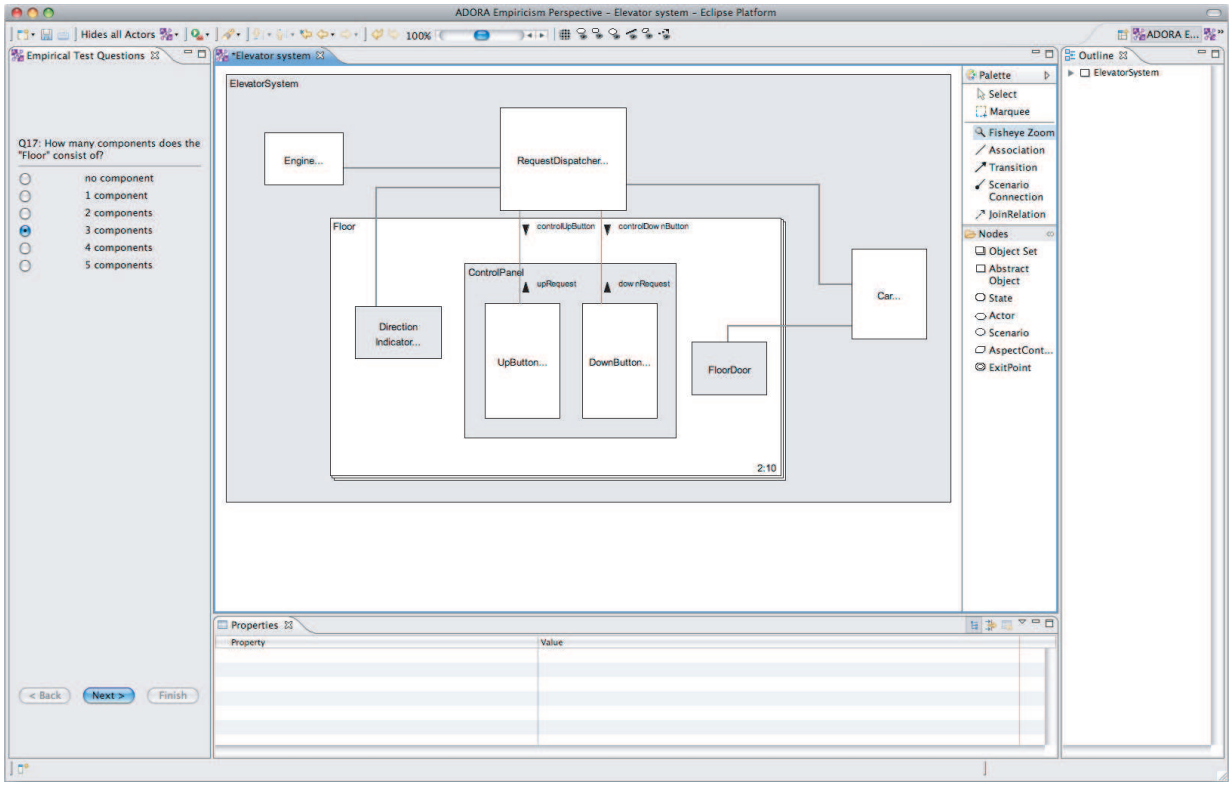


Figure 16.1: Screenshot of the empirical testing environment

Organization

The experiment started with a 25 minute introduction of the relevant constructs of the ADORA language, the ADORA tool as well as the experiment itself. The subjects then started the experiment by answering two introductory questions: their gender and their modeling skills. Subsequently, they had to answer a first set of 15 questions about the mine drainage system. The first group had to use the fisheye zoom while answering these questions, while the second group (i.e., the control group) had to answer the questions using the fully expanded model. A set of 15 questions about the elevator system followed with the roles of the two groups exchanged. The first group was now the control group using the fully expanded model while the second group had to use the fisheye zoom. At the end both groups had to subjectively judge the usefulness of the fisheye zoom and an optional linear zoom in answering the questions. The answers and the time required were gathered automatically for all subjects. The subjects required between 22 and 45 minutes to answer the 34 questions.

Subjects

We conducted the experiment with eight (one female and seven male) students simultaneously at one date as part of a requirements engineering master course at the University of Zurich. The subjects had no or only little experience with the ADORA language. Seven subjects rated their general modeling skills in a self-assessment as medium (i.e., selected the answer "I use modeling techniques from time to time.") while one assessed his/her modeling skills as high (i.e., selected the answer "I am experienced."). As a preliminary preparation they had to read the papers [Glinz et al., 2002, 2007; Meier et al., 2006] and to model a small case study [Davies et al., 2001] in the ADORA language using the ADORA tool. Before the actual experiment started, they were additionally given a short introduction on those parts of the language and the tool which were relevant for the experiment. The subjects were then randomly and equally distributed among two groups. The experiment was conducted anonymously in order to avoid biased answers.

16.2 Results

The raw results (i.e., correctness of the answers and time needed) for each individual subject can be found in tables A.1 and A.2 in the appendix. Fig. 16.2 and 16.3 show the resulting performance points as box plots for the mine drainage and the elevator system. The gray boxes show the results for the group that used the fisheye zoom while the white boxes show the performance of the control group. A direct comparison makes only sense within a question and not between questions because there were different kinds of questions (i.e., employ different tasks which has an influence on the results [Büring et al., 2006; Gutwin and Fedak, 2004]). For example, the time required to answer question Q15 which involved the examination of the interplay between multiple components and connections was significantly larger than the one to answer question Q6 (a simple listing of all components contained within a specific component). This results in large differences in the achieved performance.

Statistical Analysis

We conducted a simple unpaired[2] t-test with an error probability of 5% to check whether the mean of the subjects that used the fisheye zoom is distinct from the one of the subjects that worked with the fully expanded model. We applied the test to each question separately to find out whether the outcome depends on the kind of task that a specific question employs. The results of the statistical analysis can be found in tables A.3 and A.4 in the appendix. The only statistically significant difference between using the fisheye zoom and working on a fully expanded model occurs in question Q20 concerning the elevator system (i.e., we could only reject our null hypothesis for

[2]We actually have paired questions for the mine drainage and elevator system (e.g., questions Q3 and Q18 demand the same task) but cannot use a paired t-test since the models are not the same.

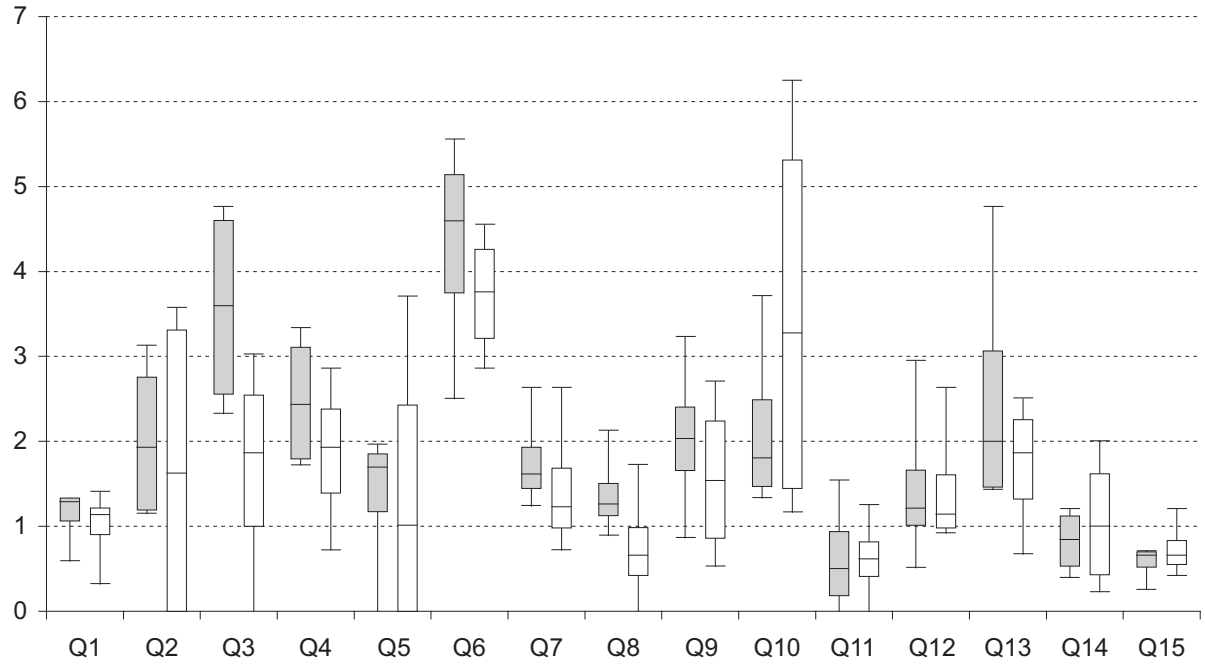


Figure 16.2: Performance points for the mine drainage system

question Q20). The control group performed significantly better than the group using the fisheye zoom while answering this specific question. The subjects had to search the whole model for a specific node and indicate the component this state lies in. The group using the fisheye zoom had to open (and close) a large fraction of all nodes in the model which gives an explanation for its bad performance (see also the detailed analysis of the interaction below). Interestingly, no significant difference has been found for Q5, the equivalent question for the mine drainage system.

Subjective Questions

At the end of the experiment the subjects were asked in two questions (Q31 and Q32) to subjectively judge the usefulness of the fisheye zoom and a traditional global (linear) zoom in answering the objective questions. Six of them stated that the fisheye zoom makes it easier to answer the questions while two selected the answer that it makes it easier in some cases. The answers were more diverse for the global zoom: Half of the subjects did not use it at all (i.e., they navigated the fully expanded model by panning only). Two selected the answer that the global zoom makes it easier in some cases to answer the questions, one that it does not have any influence and one that it makes it more difficult to answer the questions.

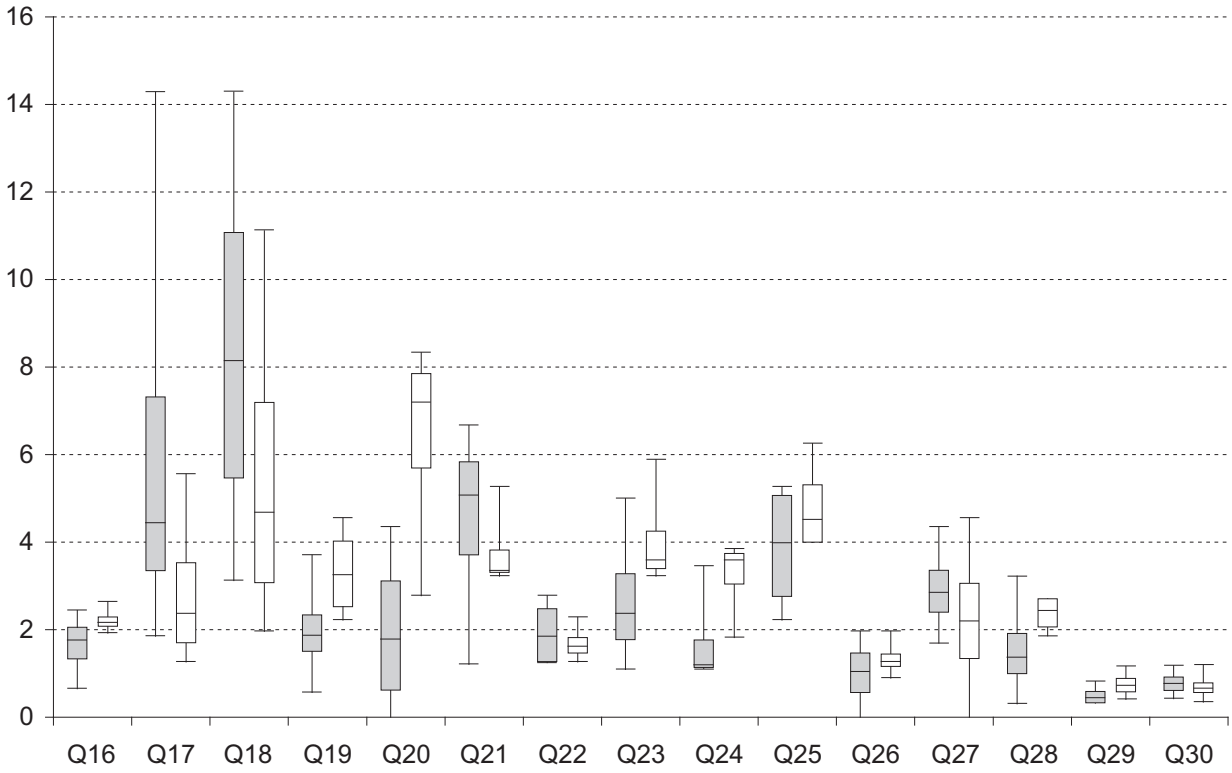


Figure 16.3: Performance points for the elevator system

Analysis of the Subjects' Interaction

The automatic recording of all zoom operations (together with a time stamp) facilitates a detailed analysis of how exactly the subjects used the fisheye zoom. Additionally, it helps in understanding and explaining the results of the statistical analysis. Fig. 16.4 shows a graphical representation of the interaction of one of the subjects with the mine drainage system. The progress of time in seconds is shown on the horizontal axis. The dashed vertical lines stand for the points in time when the subject started the next question (i.e., finished answering the previous question). The zoom-able nodes of the model, which are the nodes that have at least one contained node so that they can be zoomed-in and -out, are depicted on the vertical axis. The nodes are grouped by the hierarchical level with the whole mine drainage control system (MDCS) at the top. Each dot in the plot area stands for the execution of a zoom operation on the according node. Each line between two dots shows the time interval the node has been zoomed in (i.e., its children have been visible). A dotted line indicates that a node has not explicitly been zoomed-out by the user but its internals and the node itself are not visible because one of its parent nodes has been zoomed out.

Fig. 16.4 gives an explanation for the problems that arise if the fisheye zoom approach is used for a task that employs a general search of the model. The answering of question Q5 which is

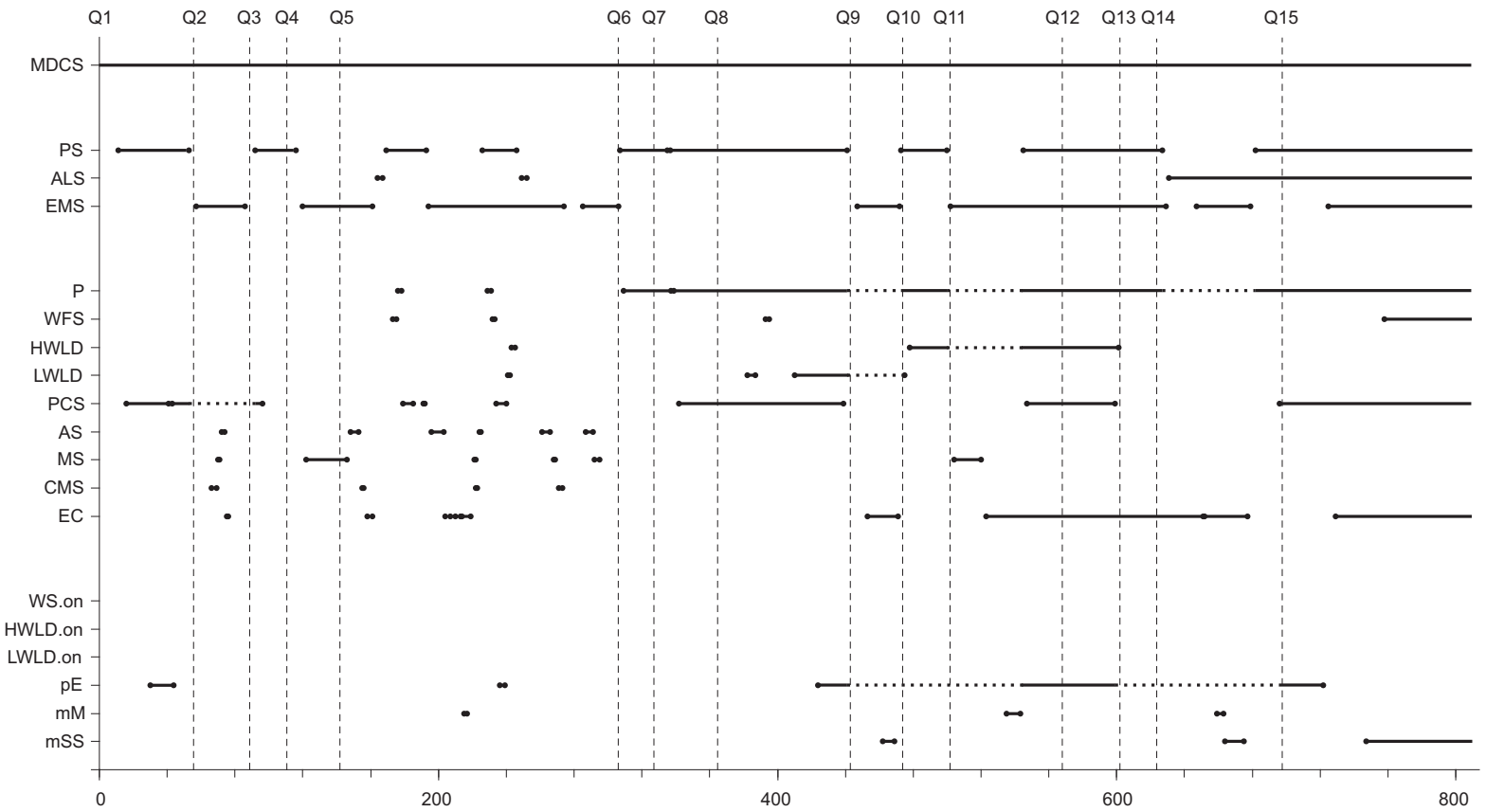


Figure 16.4: Interaction of Subject 1 with the mine drainage system

analogous to question Q20 requires a large set of zoom operations. All nodes except for four on the lowest hierarchical level are zoomed-in and -out up to five times. This explains the bad performance of the fisheye group for this kind of question. But on the other hand, the figure also shows that the subjects use both the zoom-in as well as the zoom-out facility frequently. Before the experiment we speculated that subjects may only use the zoom-in operation and never close the nodes again so that the model degenerates over time into a fully expanded model. However, this was definitely not the case as zoom-in and -out operations roughly balance each other. Furthermore, the recorded interaction of Fig. 16.4 shows that the subject closes the nodes most of the time in the same order as they were opened (i.e., it only rarely takes a shortcut by directly closing one of the parent nodes). The interaction patterns look similar to the one in Fig. 16.4 for the other subjects and the other case study.

Discussion of the Results

For most of the task employed by our experiment there was no significant difference in performance between the group using the fisheye zoom and the control group working on a fully expanded model. The interaction overhead of the fisheye zoom that was caused by the need to continuously open and close nodes seems to have been compensated by the time savings resulting from interacting with a smaller and simpler model. However, working with bigger models may result in different (and clearer) results. The only real weakness of our fisheye approach compared to a fully expanded model revealed by the experiment are undirected searches over the whole model (i.e., looking for a specific node in a large model without any further indications than its name). For this task the ability to scan big parts of the model at once without the need for interactions has emerged as a big advantage resulting in significant better performance. On the other hand, the recorded interaction patterns show that the subjects used the possibilities offered by the fisheye zoom more intensively than initially expected.

16.3 Validity of the Experiment

As any empirical work, the design of the presented controlled experiment has some potential weaknesses. The following list gives a short overview over some of them and discusses the attempts that have been made to mitigate these threats to the validity of the experiment.

- The self-assessments of modeling skills were more or less the same for all subjects, but there were often huge variances in the measured performance within a group. Thus the **individual modeling skills** may have large influence on the result and the large variances within the fisheye zoom and control group makes it hard to find any significant differences in the between group variances.
- The **sample size** was with only eight subjects very small. This may result in statistical outliers as the sample of subjects may not be representative for the population of potential

users. Additionally, such a small sample size makes it hard to find any significant differences. The small number of subjects resulted from practical considerations as the course the subjects were recruited from did not have more participants.

- Deciding on the **size and complexity of the case studies** results in a trade-off because the models should be simple enough to be understandable within a narrow time frame but, at the same time, they should have a minimal size and complexity since we are addressing the size and complexity problem. The models of the two case studies were with roughly 60 nodes and 60 links rather small so that the results may not apply to models with a realistic size of several hundred nodes and links.

- The **questions** that are asked during the experiment must be realistic and relevant for graphical models. At the same time the influence of the notation and its semantics on the results must be minimized. Therefore, the questions were restricted to the general semantics of nested node-link diagrams (i.e., connectedness, containment), which are relevant for almost all graphical models, together with a subset of the statechart semantics (i.e., states, events and actions) as an example of facts that are partially encoded in the graphical representation and partially in the semantics of the language. Furthermore, the questions were of various levels of difficulty.

- In order to exclude confounding factors concerning the **notation used for the models** and the employed semantics, we restricted the used notation to a simpler subset of the ADORA language consisting of concepts that the subjects were already familiar with (general node-link diagrams and the statechart semantics). Concerns about the influence of the notation on the results were also one of the main reasons why we decided against a comparison of the fisheye and an explosive zoom because showing the models with an explosive zoom would have entailed severe restrictions of the notation or the fisheye zoom (e.g., links that cross the borders of a parent node are incompatible with the concept of an explosive zoom).

- The **order of the questions** was not considered during the experiment even though it may have a big impact on the results because the performance may improve over time as the subjects build up their mental map. Additionally, learning effects may result in a better performance over time so that the results are better for the second case study. To mitigate the influence of such learning effects we compared the performance of the fisheye zoom group to the one of the control group for each case study (or more precisely each question) individually and did not do any comparisons of the results from different case studies.

- We used the same **tool as test environment** for both the fisheye zoom and control group with just the fisheye zoom being turned off for the control group to minimize its influence on the results. Additionally, the results were gathered automatically and anonymously to avoid any bias.

Part V

Conclusions

CHAPTER 17

Conclusions and Future Work

17.1 Summary and Achievements

Graphical representations are omnipresent in the software engineering field, but most current graphical modeling languages do not scale with the increasing size and complexity of today's systems. The navigation in the diagrams becomes a major problem especially if different aspects of the system are scattered over multiple, only loosely coupled diagrams, as it is the case with most state of the art modeling languages and tools. In this thesis, we have presented the basic algorithms for an alternative approach based on an integrated hierarchical model and tool support for the visualization and navigation. The user can use a fisheye zoom to navigate in the hierarchy and to reduce the complexity of the diagrams by hiding currently irrelevant details. A view generation mechanism that shows different aspects of the system in one diagram offers an additional dimension to reduce a diagram's complexity.

As a summary of the work, we can answer the research questions of Section 1.2 as follows:

- *What are the basic requirements for algorithms enabling a tool-supported complexity management technique? How do these algorithms look like?*

  Besides the fundamental requirements of producing a compact layout and avoiding overlaps between nodes, the preservation of the user's mental map and the stability of the layout play a crucial role. The importance of how the user perceives and especially recognizes a specific layout was the most important lessons we learned from using our previous ADORA tool implementation. As a consequence and in contrast to other approaches, our fisheye

zoom algorithm guarantees stable layouts independent of the order of zooming-out and zooming-in operations, supports model editing and runs in linear time. The importance of this layout stability was the main reason for using an interval scaling approach instead of one based on translation vectors or heuristics. An additional problem with most existing zoom or layout adaption techniques is that they do not support or not even permit editing operations. However, contrary to pure visualization tools, editing is one of the basic functionalities of a modeling tool.

- *Which additional modeling activities have to be supported or automated by a modeling tool? What are the properties such supporting or automating techniques must have and how can the underlying algorithms look like?*

  The nested box notation used to depict hierarchical relations makes tool support for editing operations essential because a change in one part of the model is propagated all the way up the hierarchy. The user has to manually adjust the size of all direct and indirect parent nodes and the location of their siblings. Thus, we have extended our fisheye zoom algorithm so that it can be used to adjust the layout automatically if a node is inserted, moved or removed.

  The drawing of the lines connecting the nodes and the placement of the labels accompanying these lines are other activities that burden the user with a lot of tedious manual work in most current modeling tools. This problem worsens with the fisheye zoom and the generation of different views because these techniques result in a dynamic layout that changes constantly. This has a big impact on the lines because they have to be adjusted after each layout change. Thus, we have also developed an algorithm for line routing in such hierarchical and dynamical models. In particular, our algorithm produces an esthetically appealing layout, routes in real-time, and preserves the secondary notation of the diagrams as far as possible. Additionally, we have developed a basic label placement technique to adjust the position and size of line labels automatically if the line and/or layout changes. While these algorithms are an essential part of interactive and dynamic models, they can also be used independently of the zoom algorithm on ordinary (flat) diagrams.

- *How does the interaction with such a modeling tool look like and are the proposed concepts really useful?*

  Our controlled experiment has shown that for most of the task employed in the experiment there was no significant difference in performance between the group using the fisheye zoom and the control group working on a fully expanded model. The interaction overhead of the fisheye zoom that was caused by the need to continuously open and close nodes seems to have been compensated by the time savings resulting from interacting with a smaller and simpler model. However, the models used for the experiment were with around 60 nodes and links rather small and the results may look different for larger models.

Thus, we have shown that it is possible to realize a visualization and navigation technique based on a fisheye zoom and the generation of dynamic views on the fly. The focus of our work

lies on the technical solutions required by an existing visualization concept [Berner, 2002] and we did not investigate whether an integrated model with dynamic views is superior to other approaches (e.g., horizontal and/or vertical abstraction over multiple separate diagrams). We have demonstrated our concepts in the context of the ADORA language and its accompanying tool (because the filtering mechanism plays to its strength only with a language employing an integrated model). However, the presented concepts can be used for any graphical modeling language that supports a hierarchical decomposition (e.g., hierarchical UML diagrams such as component diagrams).

17.2 Limitations

While our zoom algorithm permits and even supports the editing of a model, it can not completely resolve the basic trade-off between having an editable layout and the stability of this layout. If a user changes a diagram he expects that the layout changes too (e.g., that existing nodes are moved away to provide the space required by a new node). However, if some of the model elements are not visible in the diagram that is edited by the user (because some nodes are zoomed out or not relevant for the actual view) these layout changes are sometimes not directly visible. Instead they show up later after some zoom operations or the selection of another view. As the consequences of an edit operation are not directly visible, the user can sometimes not reproduce and understand some layout changes.

Another basic problem that remains with big models is the fact that even with our fisheye view concept they frequently grow beyond the size of the screen. Therefore we have to provide additionally scroll bars and a separate operation for linear view scaling when a model grows beyond the size of the available display area.

17.3 Future Work

The work presented in this thesis provides the groundwork or infrastructure to dynamically generate views by hiding nodes which facilitates a multitude of applications built on top of it. Thus the fisheye zooming concept and the generation of different views by hiding specific types of nodes are just two examples of how the concept can be used. Expanding the underlying semantics beyond those of simple node-link diagrams makes it possible to use the presented techniques for a large number of other concepts such as truly semantic fisheye views which automatically show only the relevant model elements.

But also the presented concepts give some leeway for improvements and extensions. One such extension is the creation of more compact layouts by further reducing the whitespace between nodes. This can be achieved by scaling not only the node intervals but also the neighbored space intervals. However, this easily results in a trade-off between a dense layout and additional white

space because the white space may be useful (e.g., for link labels). Another remaining problem is the stability of the layout in the case of editing operations. The stability can be further increased by adjusting the zoom holes instead of just resetting them during editing operations.

Currently, our line routing algorithm is limited to rectilinear routing. This is mainly due to the fact that the tool's underlying framework already provides a direct router. Splines may be an alternative. However, determining intersections is more complex for splines. Additionally, the presented approach to place the labels which accompany lines is currently rather a proof of concept than a mature implementation. While the current implementation shows that the concepts works, it can and has to be improved and extended in many places.

Part VI

Appendix

APPENDIX A

Details of the Experimental Validation

The details about the experimental validation are presented in the following sections. Their explanation and interpretation can be found in Chapter 16.

A.1 Case Studies

The *mine drainage system* concerns the software necessary to manage a simplified pump control system for a mining environment. The pump is used to pump mine water, which collects at the bottom of a shaft, to the surface. The main safety requirement is that the pump must not operate when the level of methane gas in the mine reaches a critical value due to the risk of explosion. Additionally, the system must measure the level of carbon dioxide in the air and detect whether there is an adequate flow of air. An alarm must be activated if the carbon dioxide level or airflow become critical. The mine drainage system case study is widely used in the software engineering field and goes back to Kramer et al. [1983]. The model is shown in Fig. A.1. It consists of 55 nodes (14 components and 41 states) and 63 links (13 associations and 50 transitions).

The second case study, the *elevator system* (which was originally introduced by Filman and Friedman [1984]), comprises the control software of a simple single car (cabin) elevator system. Parts of the system are the different operation units in the car and on the floor as well the control of the door mechanism and the motor. Fig. A.2 shows the model of the elevator system that was used for the experiment. The model of the elevator system contains 67 nodes (33 components and 34 states) and 63 links (19 associations and 44 transitions).

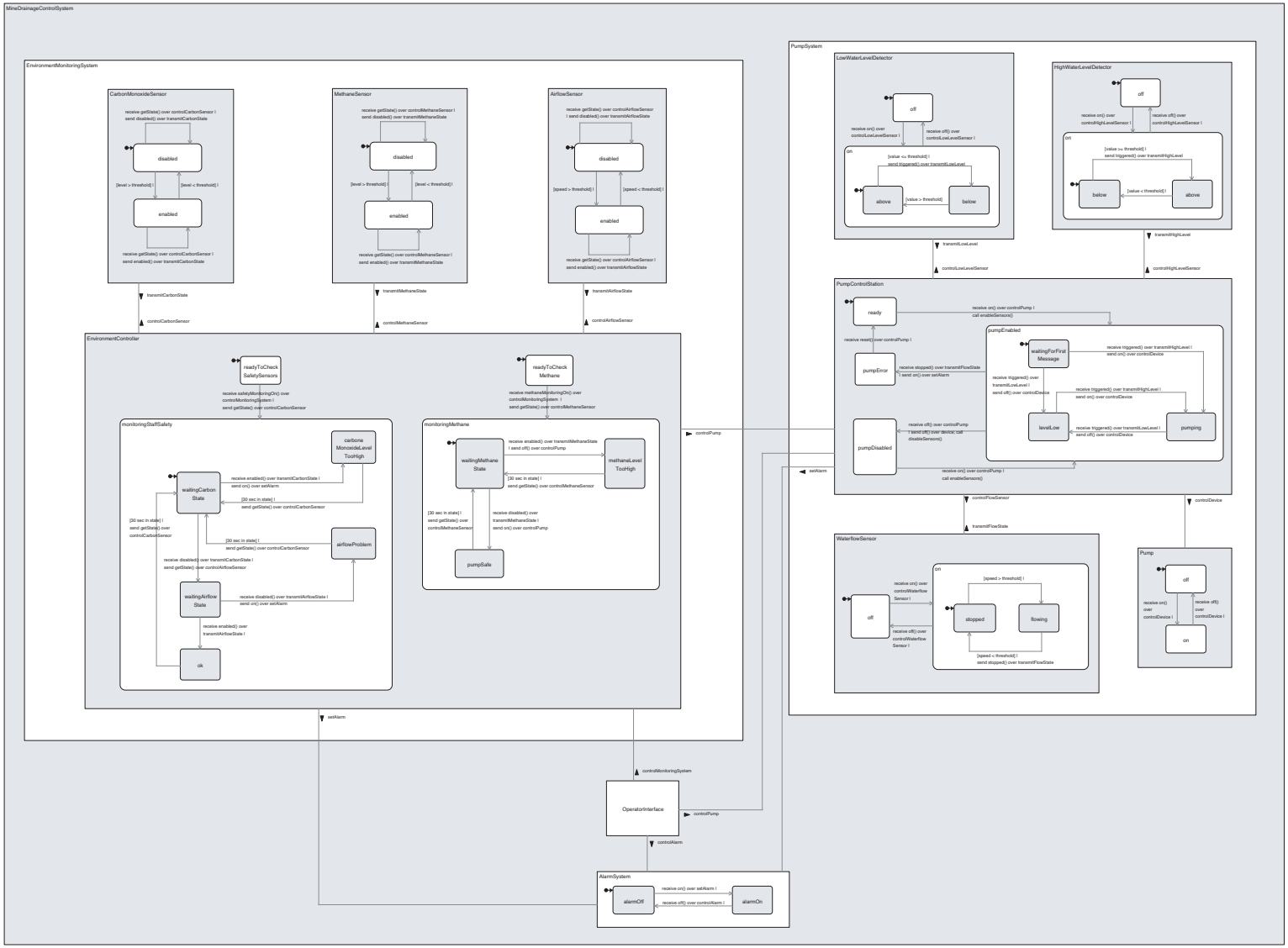


Figure A.1: Mine drainage system

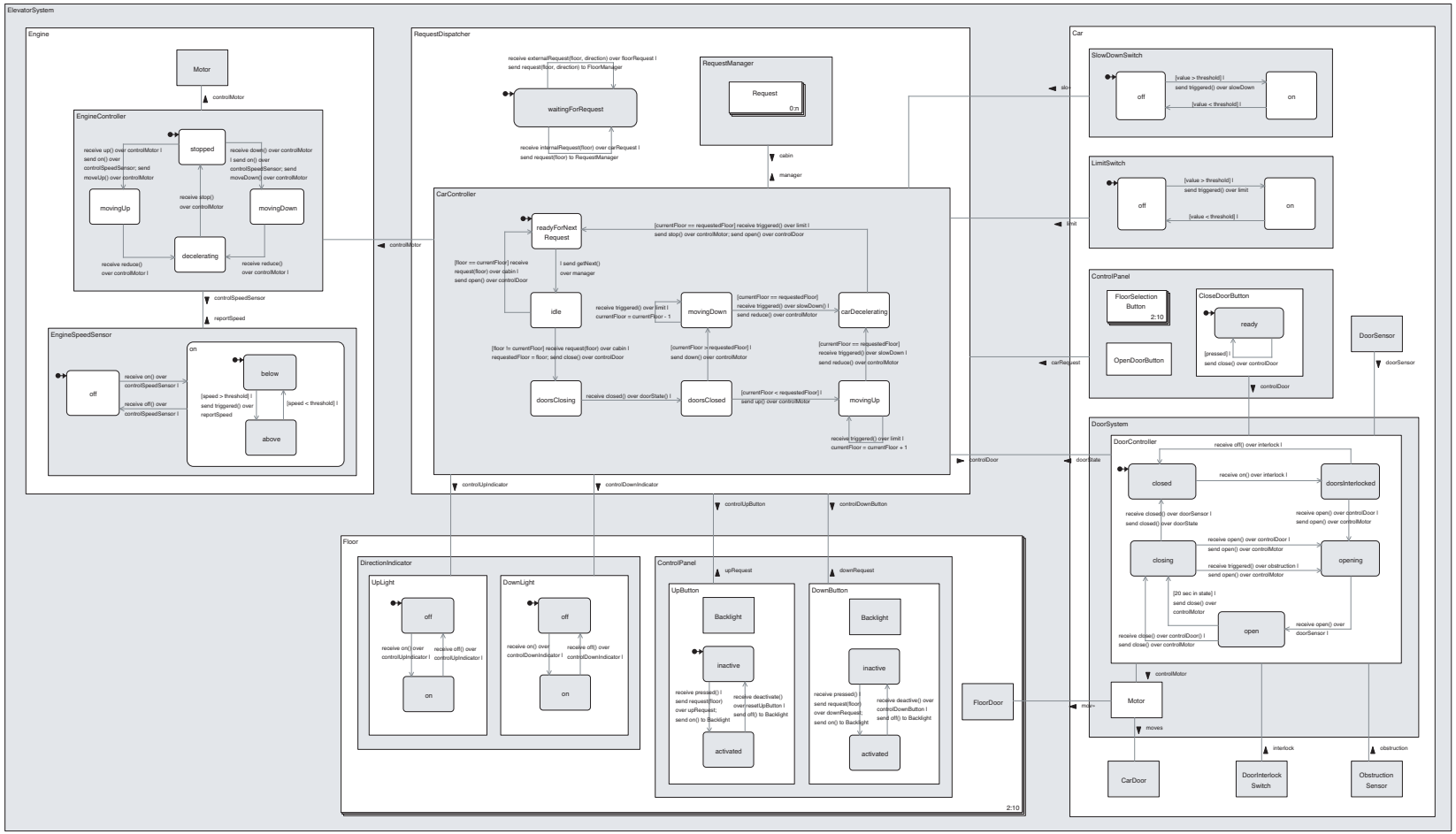


Figure A.2: Elevator system

A.2 Questionnaires

The questionnaires incorporate three kinds of questions: First, the subjects where asked about their gender and for a self-assessment of their modeling skills. Second, they had to answer a set of objective questions about the model of the mine drainage and elevator system respectively. And third, the subjects had to judge subjectively the usefulness of the fisheye and a linear zoom.

A.2.1 Personal Information

PQ1: Are you female or male?

- ○ female
- ○ male

PQ2: How experienced are you in modeling of software systems?

- ○ I am not knowledgeable in modeling software systems.
- ○ I have theoretical knowledge, but I don't practice it.
- ○ I use modeling techniques from time to time.
- ○ I am experienced.
- ○ I am an expert.

A.2.2 Mine Drainage System

Q1: Indicate the states that are embedded in the "PumpControlStation" component:

- • ready
- ○ off
- • pumpEnabled
- ○ waiting
- • pumping

Q2: How many components does the "EnvironmentMonitoringSystem" consist of?

- ○ no component
- ○ 1 component
- ○ 2 components
- ○ 3 components
- • 4 components
- ○ 5 components

Q3: Is the “OperatorInterface” component connected to the “PumpControlStation” component?

- ● Yes
- ○ No

Q4: Select the description of the transition from the state “enabled” to the state “disabled” of the “MethaneSensor” component:

- ○ [speed < threshold] |
- ○ receive on() over controlMethaneSensor |
- ● [level < threshold] |
- ○ [30 sec in state] | send getState() over controlMethaneSensor

Q5: Indicate the name of the component the state “airflowProblem” lies in:

- ○ WaterflowSensor
- ○ AirflowSensor
- ○ PumpControlStation
- ● EnvironmentController
- ○ Pump

Q6: Which of the following states does the component “Pump” consist of?

- ○ stopped
- ● off
- ● on
- ○ pumping
- ○ disabled

Q7: Select the events that cause the system to leave the state “pumpEnabled”:

- ○ on()
- ○ disable()
- ● off()
- ● stopped()
- ○ error()

Q8: Indicate the association role the event “triggered()” has to be sent over to change the state of the “PumpControlStation” component from “levelLow” to “pumping”.

- ● transmitHighLevel
- ○ controlPump
- ○ transmitFlowState

- ○ transmitLowLevel
- ○ controlDevice

Q9: The component “EnvironmentController” is initially in the state “readyToCheckSafetySensors”. Which state is active after the event “safetyMonitoringOn()” is received?

- ○ airflowProblem
- ○ monitoringMethane
- ○ ok
- ● waitingCarbonState
- ○ enabled

Q10: Which event sets the state of the “HighWaterLevelDetector” to “on”?

- ○ activate()
- ● on()
- ○ enable()
- ○ off()
- ○ start()

Q11: The component “EnvironmentController” is in the state “waitingMethaneState” and the “PumpControlStation” in the state “pumping”. Indicate the state of the “PumpControlStation” component after the “EnvironmentController” component receives the event “enabled()”.

- ○ pumpError
- ○ pumping
- ○ levelLow
- ● pumpDisabled
- ○ methaneLevelTooHigh

Q12: Can the “PumpControlStation” component be in the following sequence of states: “ready”, “waitingForFirstMessage”, “levelLow”, “pumpDisabled”, “pumpError”, “ready”?

- ○ Yes
- ● No

Q13: Indicate the components that are part of the “PumpSystem” component.

- ● LowWaterLevelDetector
- ● Pump
- ○ PumpController
- ● WaterflowSensor

- ○ AirflowSensor

Q14: Select the components that can turn the alarm on by setting the state of the “AlarmSystem” to “alarmOn”?

- ○ WaterflowSensor
- ○ OperatorInterface
- ● EnvironmentController
- ● PumpControlStation
- ○ Pump

Q15: Assume that the “PumpControlStation” is in the state “pumpEnabled”. Indicate the events that are directly or indirectly generated when the “WaterflowSensor” leaves the state “flowing” because the speed falls below the threshold.

- ○ triggered()
- ● stopped()
- ○ off()
- ○ enabled()
- ● on()

A.2.3 Elevator System

Q16: Indicate the states that are embedded in the “DoorController” component:

- ● closed
- ● open
- ○ on
- ○ stopped
- ● closing

Q17: How many components does the “Floor” consist of?

- ○ no component
- ○ 1 component
- ○ 2 components
- ● 3 components
- ○ 4 components
- ○ 5 components

Q18: Is the “Floor” component connected to the “Engine” component?

- ○ Yes
- ● No

Q19: Select the description of the transition from the state“movingDown” to the state "decelerating" of the “EngineController” component:

- ● receive reduce() over controlMotor |
- ○ receive open() over doorState |
- ○ receive down() over controlMotor |
- ○ [value > threshold] | send triggered() over limit

Q20: Indicate the name of the component the state “idle” lies in:

- ○ RequestManager
- ○ DoorSystem
- ○ EngineController
- ● CarController
- ○ CarDoor

Q21: Which of the following states does the component “LimitSwitch” consist of?

- ○ ready
- ● off
- ● on
- ○ waiting
- ○ disabled

Q22: Select the events that cause the system to leave the state “closing”:

- ● open()
- ○ triggered()
- ● closed()
- ○ on()
- ○ moving()

Q23: Indicate the association role the event “on()” has to be sent over to change the state of the “DoorController” component from “closed” to “doorsInterlocked”.

- ○ obstruction
- ○ controlDoor
- ● interlock
- ○ doorSensor

- ○ doorState

Q24: Assume that the component “CarController” is in the state “doorsClosing”. Which state is active after the event “closed()” has been received?

- ○ idle
- ○ movingUp
- ○ movingDown
- ● doorsClosed
- ○ doorsClosing

Q25: Which event sets the state of the “EngineSpeedSensor” to “on”?

- ○ activate()
- ● on()
- ○ enable()
- ○ off()
- ○ start()

Q26: The component “DoorController” is in the state “closing” and the “CarController” in the state “doorsClosing”. Indicate the state of the “CarController” component after the “DoorController” component receives the event “closed()”.

- ○ idle
- ○ closed
- ● doorsClosed
- ○ waiting
- ○ movingUp

Q27: Can the “DoorController” component be in the following sequence of states: “closed”, “doorsInterlocked”, “opening”, “closing”, “opening”, “open”?

- ○ Yes
- ● No

Q28: Indicate the components that are part of the “Car” component.

- ● CarDoor
- ● ObstructionSensor
- ○ Motor
- ● LimitSwitch
- ○ CarController

Q29: Select the components that can close the door by setting the state of the “DoorController” to “closing”?

- ● CarController
- ○ CarDoor
- ● CloseDoorButton
- ○ DoorSensor
- ○ DoorSystem

Q30: Assume that the “EngineController” is in the state “stopped”. Indicate the events that are directly or indirectly generated when the “CarController” leaves the state “doorsClosed” because the current floor is above the requested floor.

- ○ up()
- ● down()
- ○ reduce()
- ● moveDown()
- ● on()

A.2.4 Subjective Questions

Q31: How did the fisheye zoom help in answering the questions?

- ○ Makes it easier to answer the questions.
- ○ In some cases, makes it easier to answer the questions.
- ○ Does not have any influence.
- ○ Makes it more difficult to answer the questions.

Q32: How did the global scaling (linear zoom) help in answering in questions:

- ○ I did not use the global scaling.
- ○ Makes it easier to answer the questions.
- ○ In some cases, makes it easier to answer the questions.
- ○ Does not have any influence.
- ○ Makes it more difficult to answer the questions.

A.3 Raw Results

Tables A.1 and A.2 show the raw results of the experiment. The results of each subject are shown in two lines. The first line shows the proportion of correct answers for each question. The second line shows the time in seconds required to answer the question.

Table A.1: Results for the mine drainage system

	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15
Subject 1	1	1	1	1	0	1	1	1	1	1	1	1	1	0.8	0.8
	56	32	22	30	164	18	38	77	31	27	65	34	21	73	113
Subject 2	0.8	1	1	1	1	1	1	1	1	1	0	1	1	1	0.8
	60	83	21	33	55	20	66	47	47	66	155	85	70	173	311
Subject 3	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	82	38	38	55	51	24	59	83	52	48	136	81	40	252	165
Subject 4	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	169	87	43	58	64	40	81	112	116	75	406	195	86	83	144
Subject 5	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
	71	39	42	35	27	35	38	58	37	20	80	38	40	50	83
Subject 6	1	1	1	1	0	1	0.8	1	1	1	1	1	1	1	0.8
	91	31	33	45	76	24	75	135	103	65	149	109	65	202	189
Subject 7	1	1	1	1	1	1	1	0	1	1	0	1	1	1	0.8
	87	28	75	62	50	22	73	55	48	16	80	79	46	67	113
Subject 8	0.8	0	0	1	0	1	1	1	1	1	1	1	1	0.4	0.8
	250	266	89	138	240	30	139	177	190	86	182	100	149	173	135

Table A.2: Results for the elevator system

	Q16	Q17	Q18	Q19	Q20	Q21	Q22	Q23	Q24	Q25	Q26	Q27	Q28	Q29	Q30
Subject 5	1	1	1	1	1	1	1	1	1	1	1	1	1	0.8	1
	41	20	7	27	37	22	36	20	29	19	51	23	31	98	85
Subject 6	1	1	1	1	1	1	1	1	1	1	1	1	0.8	0.4	0.6
	64	26	16	55	121	18	42	37	87	20	77	33	54	124	89
Subject 7	1	1	1	1	1	1	1	1	1	1	1	1	0.6	0.6	1
	52	7	10	53	23	15	78	50	90	34	133	38	49	118	120
Subject 8	0.8	1	1	1	0	1	1	1	1	1	0	1	0.2	0.4	0.4
	123	54	32	176	60	83	80	92	83	45	60	59	65	119	94
Subject 1	1	1	1	1	1	1	0.8	1	1	1	1	1	1	0.8	1
	38	18	9	22	13	19	35	17	26	16	51	22	37	69	84
Subject 2	1	1	1	1	1	1	0.8	1	1	1	1	0	1	0.8	0.6
	47	79	29	26	36	30	48	27	55	20	80	32	37	196	93
Subject 3	1	1	1	1	1	1	0.6	1	1	1	1	1	0.8	0.6	0.8
	46	35	17	38	12	31	52	31	27	25	79	56	43	94	231
Subject 4	1	1	1	1	1	1	1	1	1	1	1	1	1	0.8	0.8
	52	54	51	45	15	30	79	29	29	25	112	39	47	101	125

A.4 Statistical Analysis

Tables A.3 and A.4 contain the statistical data for each question. Only Q20 revealed a statistically significant difference (with an error probability of 5%) between using the fisheye zoom and working on a fully expanded model.

Table A.3: Statistical data for the mine drainage system

Question	Fisheye		Expanded		t	p
	Mean	SD	Mean	SD		
Q1	1.1194	0.3559	0.9941	0.4694	0.4201	0.6854
Q2	2.0277	1.0028	1.6993	1.9672	0.2941	0.7762
Q3	3.5661	1.2650	1.6861	1.3237	1.9587	0.0858
Q4	2.4764	0.8246	1.8542	0.9083	0.9935	0.3495
Q5	1.3353	0.9053	1.4259	1.7873	0.0895	0.9309
Q6	4.3055	1.3321	3.7256	0.7691	0.7418	0.4794
Q7	1.7690	0.6054	1.4468	0.8333	0.6166	0.5546
Q8	1.3810	0.5271	0.7574	0.7177	1.3599	0.2109
Q9	2.0346	0.9686	1.5708	0.9991	0.6567	0.5298
Q10	2.1588	1.0782	3.4878	2.5246	0.9491	0.3704
Q11	0.6300	0.6783	0.6176	0.5127	0.0288	0.9777
Q12	1.4662	1.0363	1.4537	0.7991	0.0190	0.9853
Q13	2.5402	1.5617	1.7208	0.8057	0.9148	0.3871
Q14	0.8188	0.3923	1.0547	0.8319	0.5060	0.6265
Q15	0.5664	0.2110	0.7321	0.3360	0.8208	0.4356

Table A.4: Statistical data for the elevator system

Question	Fisheye		Expanded		t	p
	Mean	SD	Mean	SD		
Q16	1.6437	0.7536	2.2140	0.2989	1.3660	0.2091
Q17	6.2459	5.5153	2.8825	1.8992	1.1263	0.2927
Q18	8.4151	4.8183	5.6006	4.0135	0.8811	0.4040
Q19	1.9942	1.2907	3.3113	1.0735	1.5172	0.1677
Q20	1.9692	1.9476	6.3675	2.4896	2.5948	**0.0319**
Q21	4.4931	2.3571	3.7889	0.9841	0.5440	0.6013
Q22	1.9226	0.7754	1.6891	0.4313	0.5194	0.6175
Q23	2.6974	1.6715	4.0650	1.2271	1.2835	0.2353
Q24	1.7284	1.1472	3.2040	0.9384	1.9028	0.0936
Q25	3.8566	1.5049	4.8125	1.0680	1.0142	0.3402
Q26	1.0028	0.8314	1.3423	0.4467	0.7082	0.4989
Q27	2.9261	1.1007	2.2238	1.8835	0.6345	0.5435
Q28	1.5598	1.2195	2.3483	0.4234	1.1913	0.2677
Q29	0.4958	0.2297	0.7494	0.3155	1.2649	0.2415
Q30	0.7773	0.3145	0.7054	0.3522	0.3010	0.7711

Bibliography

Appert, C. and Fekete, J.-D. (2006). OrthoZoom Scroller: 1D Multi-Scale Navigation. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 21–30.

Bartram, L., Ho, A., Dill, J., and Henigman, F. (1995). The Continuous Zoom: A Constrained Fisheye Technique for Viewing and Navigating Large Information Spaces. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology (UIST '95)*, pages 207–215.

Battista, G. D., Eades, P., Tamassia, R., and Tollis, I. (1994). Annotated Bibliography on Graph Drawing Algorithms. *Computational Geometry: Theory and Applications*, 4:235–282.

Baudisch, P., Lee, B., and Hanna, L. (2004). Fishnet, A Fisheye Web Browser with Search Term Popouts: A Comparative Evaluation with Overview and Linear View. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 133–140.

Bederson, B. B. (2000). Fisheye Menus. In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology*.

Bederson, B. B. and Boltman, A. (1999). Does Animation Help Users Build Mental Maps of Spatial Information? In *Proceedings of the 1999 IEEE Symposium on Information Visualization (InfoVis'99)*, pages 28–35.

Bederson, B. B., Clamage, A., Czerwinski, M. P., and Robertson, G. G. (2004a). DateLens: A Fisheye Calendar Interface for PDAs. *ACM Transactions on Computer-Human Interaction*, 11(1):90–119.

Bederson, B. B., Grosjean, J., and Meyer, J. (2004b). Toolkit Design for Interactive Structured Graphics. *IEEE Transactions on Software Engineering*, 30(8):535–546.

Bederson, B. B. and Hollan, J. D. (1994). Pad++: A Zooming Graphical Interface for Exploring Alternate Interface Physics. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology*, pages 17–26.

Bederson, B. B., Meyer, J., and Good, L. (2000). Jazz: An Extensible Zoomable User Interface Graphics Toolkit in Java. In *Proceedings of the 13th Annual ACM Symposium on User Interface Software and Technology*, pages 171–180.

Berner, S. (2002). *Modellvisualisierung für die Spezifikationssprache ADORA*. PhD thesis, Universität Zürich.

Berner, S., Joos, S., Glinz, M., and Arnold, M. (1998). A Visualization Concept for Hierarchical Object Models. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering (ASE'98)*, pages 225–228.

Booth, T. (1967). *Sequential Machines and Automata Theory*. John Wiley and Sons, New York.

Bourgeois, F. and Guiard, Y. (2002). Multiscale Pointing: Facilitating Pan-Zoom Coordination. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'02)*, pages 758–759.

Bridgeman, S. and Tamassia, R. (2000). Difference Metrics for Interactive Orthogonal Graph Drawing Algorithms. *Journal of Graph Algorithms and Applications*, 4(3):47–74.

Brooks, F. P. (1987). No Silver Bullet - Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19.

Brooks, F. P. (1995). *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 2nd edition.

Büring, T., Gerken, J., and Reiterer, H. (2006). User Interaction with Scatterplots on Small Screens - A Comparative Evaluation of Geometric-Semantic Zoom and Fisheye Distortion. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):829–836.

Card, S. K., Robertson, G. G., and Mackinlay, J. D. (1991). The Information Visualizer, an Information Workspace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Reaching Through Technology*, pages 181–186.

Carpendale, M. S. T., Cowperthwaite, D. J., and Fracchia, F. D. (1997a). Extending Distortion Viewing from 2D to 3D. *IEEE Computer Graphics and Applications*, 17(4):42–51.

Carpendale, M. S. T., Cowperthwaite, D. J., and Fracchia, F. D. (1997b). Making Distortions Comprehensible. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 36–45.

Chen, P. P.-S. (1976). The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36.

Chimera, R. (1992). Value Bars: An Information Visualization and Navigation Tool for Multi-attribute Listings. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 293–294.

Christensen, J., Marks, J., and Shieber, S. (1995). An Empirical Study of Algorithms for Point-Feature Label Placement. *ACM Transactions on Graphics*, 14(3):203–232.

Chui, M. and Dillon, A. (1997). Who's Zooming Whom? Attunement to Animation in the Interface. *Journal of the American Society for Information Science*, 48(11):1067–1072.

Cockburn, A., Karlson, A., and Bederson, B. B. (2008). A Review of Overview+Detail, Zooming, and Focus+Context Interfaces. *ACM Computing Surveys*, 41(1):1–31.

Cockburn, A. and Savage, J. (2003). Comparing Speed-Dependent Automatic Zooming with Traditional Scroll, Pan, and Zoom Methods. In *People and Computers XVII: British Computer Society Conference on Human Computer Interaction*, pages 87–102.

Coleman, M. K. and Parker, D. S. (1996). Aesthetics-based Graph Layout for Human Consumption. *Software: Practice and Experience*, 26(12):1415–1438.

Constantine, L. L. and Lockwood, L. A. D. (1999). *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. Addison-Wesley Professional.

Czerwinski, M. and Larson, K. (1997). The New Web Browsers: They're Cool but Are They Useful? In *People and Computers XII: British Computer Society Conference on Human Computer Interaction*.

Davies, N., Cheverst, K., Mitchell, K., and Efrat, A. (2001). Using and Determining Location in a Context-Sensitive Tour Guide. *IEEE Computer*, 34(8):35–41.

de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. (2000). *Computational Geometry: Algorithms and Applications*. Springer, Heidelberg, second edition.

DeMarco, T. (1979). *Structured Analysis and Systems Specification*. Prentice-Hall, Englewood Cliffs, N.J.

Dijkstra, E. W. (1959). A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271.

Dijkstra, E. W. (1968). Go To Statement Considered Harmful. *Communications of the ACM*, 11(3):147–148.

Dijkstra, E. W. (1989). The Cruelty of Really Teaching Computer Science. *Communications of the ACM*, 32(12):1389–1404.

Dill, J., Bartram, L., Ho, A., and Henigman, F. (1994). A Continuously Variable Zoom for Navigating Large Hierarchical Networks. In *Proceedings of IEEE International Conference on Systems, Man and Cybernetics*, pages 386–390.

Donelson, W. C. (1978). Spatial Management of Information. In *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*, pages 203 – 209.

Donoho, A. W., Donoho, D. L., and Gasko, M. (1988). MacSpin: Dynamic Graphics on a Desktop Computer. *IEEE Computer Graphics & Applications*, 8(4):51–58.

Donskoy, M. and Kaptelinin, V. (1997). Window Navigation With and Without Animation: A Comparison of Scroll Bars, Zoom, and Fisheye View. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 279–280.

Dori, D. (2002). Why Significant UML Change Is Unlikely. *Communications of the ACM*, 45(11):82–85.

Eades, P. (1984). A Heuristic for Graph Drawing. *Congressus Numerantium*, 42:149–160.

Eades, P., Lai, W., Misue, K., and Sugiyama, K. (1991). Preserving the Mental Map of a Diagram. In *Proceedings of COMPUGRAPHICS '91*, pages 34–43.

Farrand, W. A. (1973). *Information Display in Interactive Design*. PhD thesis, Department of Engineering, University of California, Los Angeles.

Filman, R. E. and Friedman, D. P. (1984). *Coordinated Computing: Tools and Techniques for Distributed Software*. McGraw Hill Higher Education.

Fish, A. and Störrle, H. (2007). Visual qualities of the Unified Modeling Language: Deficiencies and Improvements. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 41–49.

Freire, M. and Rodríguez, P. (2006). Preserving The Mental Map in Interactive Graph Interfaces. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 270–273.

Furnas, G. W. (1981). The Fisheye View: A New Look at Structured Files. Technical report, Bell Labs.

Furnas, G. W. (1986). Generalized Fisheye Views. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 16–23.

Furnas, G. W. (2006). A Fisheye Follow-up: Further Reflections on Focus + Context. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 999–1008.

Furnas, G. W. and Bederson, B. B. (1995). Space-Scale Diagrams: Understanding Multiscale Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 234 – 241.

Furnas, G. W. and Zhang, X. (1998). MuSE: A Multiscale Editor. In *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology (UIST'98)*, pages 107–116.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Gershon, N., Eick, S. G., and Card, S. (1998). Information Visualization. *interactions*, 5(2):9–15.

Glinz, M. (2000). Problems and Deficiencies of UML as a Requirements Specification Language. In *Proceedings of the 10th International Workshop on Software Specification and Design*, pages 11–22.

Glinz, M. (2002). Statecharts For Requirements Specification - As Simple As Possible, As Rich As Needed. In *Proceedings of the ICSE 2002 Workshop on Scenarios and State Machines: Models, Algorithms, and Tools*.

Glinz, M., Berner, S., and Joos, S. (2002). Object-oriented modeling with ADORA. *Information Systems*, 27(6):425–444.

Glinz, M., Seybold, C., and Meier, S. (2007). Simulation-Driven Creation, Validation and Evolution of Behavioral Requirements Models. Dagstuhl-Workshop Modellbasierte Entwicklung eingebetteter Systeme (MBEES 2007), Informatik-Bericht 2007-01, TU Braunschweig.

Goldstine, H. H. and von Neumann, J. (1947). Planning and Coding Problems for an Electronic Computing Instrument. In Taub, A. H., editor, *John von Neumann: Collected Works*, volume 5, pages 80–151. McMillan, New York.

Gonzalez, C. (1996). Does Animation in User Interfaces Improve Decision Making? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 27–34.

Gross, J. T. and Yellen, J. (1998). *Graph Theory and Its Applications*. CRC Press.

Gutwin, C. (2002). Improving Focus Targeting in Interactive Fisheye Views. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 267–274.

Gutwin, C. and Fedak, C. (2004). Interacting with Big Interfaces on Small Screens: A Comparison of Fisheye, Zoom, and Panning Techniques. In *Proceedings of Graphics Interface 2004*, pages 145–152.

Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274.

Harel, D. (1988). On Visual Formalisms. *Communications of the ACM*, 31(5):514–530.

Henderson, D. A. and Card, S. K. (1986). Rooms: The Use of Multiple Virtual Workspaces to Reduce Space Contention in a Window-Based Graphical User Interface. *ACM Transactions on Graphics*, 5(3):211–243.

Herman, I., Melançon, G., and Marshall, M. S. (2000). Graph Visualization and Navigation in Information Visualization: A Survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43.

Herot, C. F. (1980). Spatial Management of Data. *ACM Transactions on Database Systems*, 5(4):493–513.

Hornbæk, K., Bederson, B. B., and Plaisant, C. (2002). Navigation Patterns and Usability of Zoomable User Interfaces with and without an Overview. *ACM Transactions on Computer-Human Interaction*, 9(4):362–389.

Hornbæk, K. and Frøkjær, E. (2003). Reading Patterns and Usability in Visualizations of Electronic Documents. *ACM Transactions on Computer-Human Interaction*, 10(2):119–149.

Hornbæk, K. and Hertzum, M. (2007). Untangling the Usability of Fisheye Menus. *ACM Transactions on Computer-Human Interaction*, 14(2).

Huotari, J., Lyytinen, K., and Niemelä, M. (2004). Improving Graphical Information System Model Use With Elision and Connecting Lines. *ACM Transactions on Computer-Human Interaction*, 11(1):26–58.

IBM (1974). HIPO—A Design Aid and Documentation Technique. Technical Report GC20-1851, IBM Corporation, White Plains, NY.

IEEE (1990). Standard glossary of software engineering terminology. IEEE Std 610.12-1990. IEEE Computer Society Press.

Imhof, E. (1975). Positioning Names on Maps. *The American Cartographer*, 2(2):128–144.

Irani, P., Tingley, M., and Ware, C. (2001). Using Perceptual Syntax to Enhance Semantic Content in Diagrams. *IEEE Computer Graphics and Applications*, 21(5):76–84.

ITU-T (2000). Specification and Description Language (SDL). Recommendation Z.100(11/99), International Telecommunication Union, Geneva.

Jackson, M. A. (1975). *Principles of Program Design*. Academic Press.

Jakobsen, M. R. and Hornbæk, K. (2006). Evaluating a Fisheye View of Source Code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 377–386.

Joos, S. (1999). *Adora-L - Eine Modellierungssprache zur Spezifikation von Software-Anforderungen*. PhD thesis, University of Zurich.

Jul, S. and Furnas, G. W. (1998). Critical Zones in Desert Fog: Aids to Multiscale Navigation. In *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology*, pages 97–106.

Kadmon, N. and Shlomi, E. (1978). A Polyfocal Projection for Statistical Surfaces. *Cartography*, 15(1):36–41.

Kakoulis, K. G. and Tollis, I. G. (1997). An Algorithm for Labeling Edges of Hierarchical Drawings. In *Proceedings of the 5th International Symposium on Graph Drawing*, pages 169–180.

Kersten, M. and Murphy, G. C. (2005). Mylar: A Degree-of-Interest Model for IDEs. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 159–168.

Kim, J., Hahn, J., and Hahn, H. (2000). How Do We Understand a System with (So) Many Diagrams? Cognitive Integration Processes in Diagrammatic Reasoning. *Information Systems Research*, 11(3):284–303.

Klein, C. and Bederson, B. B. (2005). Benefits of Animated Scrolling. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'05)*, pages 1965–1968.

Knuth, D. (1997). *The Art of Computer Programming*, volume 1. Addison-Wesley, 3. edition.

Kobryn, C. (2004). Uml 3.0 and the future of modeling. *Software and Systems Modeling*, 13(1):4–8.

Koffka, K. (1935). *Principles of Gestalt Psychology*. Harcourt-Brace, New York.

Kramer, J., Magee, J., Sloman, M., and Lister, A. (1983). CONIC: an integrated approach to distributed computer control systems. *IEE Proceedings-E Computers and Digital Techniques*, 130(1):1–10.

Kruchten, P. (1995). The 4+1 View Model of Architecture. *IEEE Software*, 12(6):42–50.

Lamping, J., Rao, R., and Pirolli, P. (1995). A Focus+Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies. In *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'95)*, pages 401–408.

Lee, C. Y. (1961). An Algorithm for Path Connections and its Applications. *IRE Transactions on Electronic Computers*, EC-10:346–365.

Leung, Y. K. and Apperley, M. D. (1994). A Review and Taxonomy of Distortion-Oriented Presentation Techniques. *ACM Transactions on Computer-Human Interaction*, 1(2):126 – 160.

Leveson, N. G., Heimdahl, M. P. E., Hildreth, H., and Reese, J. D. (1994). Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–707.

Li, W., Eades, P., and Hong, S.-H. (2005). Navigating Software Architectures with Constant Visual Complexity. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 225 – 232.

Ludewig, J. (2003). Models in Software Engineering - An Introduction. *Software and Systems Modeling*, 2(1):5–14.

Lyons, K. A., Meijer, H., and Rappaport, D. (1998). Algorithms for Cluster Busting in Anchored Graph Drawing. *Journal of Graph Algorithms and Applications*, 2(1):1–24.

Mackinlay, J. D., Robertson, G. G., and Card, S. K. (1991). The Perspective Wall: Detail and Context Smoothly Integrated . In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 173–176.

Marple, D., Smulders, M., and Hegen, H. (1990). Tailor: A Layout System Based on Trapezoidal Corner Stitching. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9(1):66–90.

Marty, M. (2002). Analyse und Erweiterung von Algorithmen zur logischen Navigation in ADORA-Modellen. Master's thesis, Universität Zürich.

Meier, S. (2009). *Aspect-Oriented Requirements Modeling*. PhD thesis, Universität Zürich.

Meier, S., Reinhard, T., Seybold, C., and Glinz, M. (2006). Aspect-Oriented Modeling with Integrated Object Models. In *Modellierung 2006*, volume P-82 of *GI-Edition-Lecture Notes in Informatics*, pages 129–144.

Miller, G. A. (1956). The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information. *Psychological Review*, 63(2):81–97.

Miriyala, K., Hornick, S. W., and Tamassia, R. (1993). An Incremental Approach to Aesthetic Graph Layout. In *Proceeding of the Sixth International Workshop on Computer-Aided Software Engineering*, pages 297–308.

Misue, K., Eades, P., Lai, W., and Sugiyama, K. (1995). Layout Adjustment and the Mental Map. *Journal of Visual Languages and Computing*, 6(2):183–210.

Moody, D. (2006). Dealing with "Map Shock": A Systematic Approach for Managing Complexity in Requirements Modelling. In *Proceedings of the 12th Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ 2006)*.

Moody, D. L. (2009). The "Physics" of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35(6):756–779.

Mullet, K., Fry, C., and Schiano, D. (1997). On Your Marks, Get Set, Browse! In *Extended Abstracts of the ACM Conference on Human Factors in Computing Systems*, pages 113–114.

Nassi, I. and Shneiderman, B. (1973). Flowchart Techniques for Structured Programming. *ACM SIGPLAN Notices*, 8(8):12–26.

Nekrasovski, D., Bodnar, A., McGrenere, J., Guimbretière, F., and Munzner, T. (2006). An Evaluation of Pan & Zoom and Rubber Sheet Navigation with and without an Overview. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 11–20.

Noik, E. G. (1994). A Space of Presentation Emphasis Techniques for Visualizing Graphs. In *Proceedings of Graphics Interface '94*, pages 225–234.

Nordbotten, J. C. and Crosby, M. E. (1999). The Effect of Graphic Style on Data Model Interpretation. *Information Systems Journal*, 9(2):139–155.

North, S. C. (1996). Incremental Layout in DynaDAG. In *Proceedings of the 4th Symposium on Graph Drawing*, pages 409–418.

OMG (2003). MDA Guide Version 1.0.1. omg/03-06-01, Object Management Group.

OMG (2005). Unified Modeling Language: Superstructure Version 2.0. Document formal/05-07-04, Object Management Group.

Ousterhout, J. K. (1984). Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 3(1):87–100.

Parnas, D. L. (1972). On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058.

Peleg, M. and Dori, D. (2000). The Model Multiplicity Problem: Experimenting with Real-Time Specification Methods. *IEEE Transactions on Software Engineering*, 26(8):742–759.

Perlin, K. and Fox, D. (1993). Pad: An Alternative Approach to the Computer Interface. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, pages 57 – 64.

Petre, M. (1995). Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Communications of the ACM*, 38(6):33–44.

Petri, C. (1962). *Kommunikation mit Automaten*. PhD thesis, Technische Universität Darmstadt.

Pietriga, E. (2005). A Toolkit for Addressing HCI Issues in Visual Language Environments. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 145–152.

Pietriga, E., Appert, C., and Beaudouin-Lafon, M. (2007). Pointing and Beyond: An Operationalization and Preliminary Evaluation of Multi-scale Searching. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1215–1224.

Pirolli, P., Card, S. K., and Wege, M. M. V. D. (2001). Visual Information Foraging in a Focus+Context Visualization. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 506–513.

Pirolli, P., Card, S. K., and Wege, M. M. V. D. (2003). The Effects of Information Scent on Visual Search in the Hyperbolic Tree Browser. *ACM Transactions on Computer-Human Interaction*, 10(1):20–53.

Plaisant, C., Grosjean, J., and Bederson, B. B. (2002). SpaceTree: Supporting Exploration in Large Node Link Tree, Design Evolution and Empirical Evaluation. In *Proceedings of the IEEE Symposium on Information Visualization (InfoVis 2002)*, pages 57–64.

Purchase, H. C., Carrington, D., and Allder, J.-A. (2002). Empirical Evaluation of Aesthetics-based Graph Layout. *Empirical Software Engineering*, 7(3):233–255.

Rao, R. and Card, S. K. (1994). The Table Lens: Merging Graphical and Symbolic Representations in an Interactive Focus+Context Visualization for Tabular Information. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 318–322.

Reinhard, T. (2004). Halbautomatische Linienführung in Adora-Modellen unter Berücksichtigung des Zoom-Algorithmus. Master's thesis, Universität Zürich.

Reinhard, T. and Glinz, M. (2010). Automatic Placement of Link Labels in Diagrams. In *ICSE 2010 Workshop on Flexible Modeling Tools (FlexiTools 2010)*.

Reinhard, T., Meier, S., and Glinz, M. (2007). An Improved Fisheye Zoom Algorithm for Visualizing and Editing Hierarchical Models. In *Proceedings of the Second International Workshop on Requirements Engineering Visualization (REV'07)*.

Reinhard, T., Meier, S., Stoiber, R., Cramer, C., and Glinz, M. (2008). Tool Support for the Navigation in Graphical Models. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 823–826.

Reinhard, T., Seybold, C., Meier, S., Glinz, M., and Merlo-Schett, N. (2006). Human-Friendly Line Routing for Hierarchical Diagrams. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 273–276.

Riel, A. J. (1996). *Object-Oriented Design Heuristics*. Addison-Wesley.

Robertson, G. G. and Mackinlay, J. D. (1993). The Document Lens. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 101–108.

Robertson, G. G., Mackinlay, J. D., and Card, S. K. (1991). Cone Trees: Animated 3D Visualizations of Hierarchical Information. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 189–194.

Ross, D. T. and Schoman, K. E. (1977). Structured Analysis for Requirements Definition. *IEEE Transactions on Software Engineering*, 3(1):6–15.

Sarkar, M. and Brown, M. H. (1992). Graphical Fisheye Views of Graphs. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 83–91.

Sarkar, M., Snibbe, S. S., Tversky, O. J., and Reiss, S. P. (1993). Stretching the Rubber Sheet: A Metaphor for Viewing Large Layouts on Small Screens. In *ACM Symposium on User Interface Software and Technology*, pages 81–91.

Scaife, M. and Rogers, Y. (1996). External Cognition: How Do Graphical Representations Work? *International Journal of Human-Computer Studies*, 45(2):185–213.

Schaffer, D., Zuo, Z., Bartram, L., Dill, J., Dubs, S., Greenberg, S., and Roseman, M. (1993). Comparing Fisheye and Full-Zoom Techniques for Navigation of Hierarchically Clustered Networks. In *Proceedings of Graphics Interface (GI '93)*.

Schaffer, D., Zuo, Z., Greenberg, S., Bartram, L., Dill, J., Dubs, S., and Roseman, M. (1996). Navigating Hierarchically Clustered Networks through Fisheye and Full-Zoom Methods. *ACM Transactions on Computer-Human Interaction*, 3(2):162–188.

Schmidt, D. C. (2006). Model-Driven Engineering. *IEEE Computer*, 39(2):25–31.

Seybold, C. (2006). *Simulation teilformaler Anforderungsmodelle*. PhD thesis, Universität Zürich.

Seybold, C., Glinz, M., Meier, S., and Merlo-Schett, N. (2003). An Effective Layout Adaptation Technique for a Graphical Modeling Tool. In *Proceedings of the 25th International Conference on Software Engineering*, pages 826 – 827.

Shneiderman, B. (1996). The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 336–343.

Shoemaker, G. and Gutwin, C. (2007). Supporting Multi-Point Interaction in Visual Workspaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 999–1008.

Simon, H. A. (1996). *The Sciences of the Artificial*. The MIT Press, Cambridge, MA, third edition.

Sindre, G., Gulla, B., and Jokstad, H. G. (1993). Onion Graphs: Aesthetics and Layout. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, pages 287–291.

Soukup, J. (1978). Fast Maze Router. In *Proceedings of the 15th conference on Design automation*, pages 100–102.

Spence, R. (2007). *Information Visualization: Design for Interaction*. Prentice Hall, second edition.

Spence, R. and Apperley, M. (1982). Data Base Navigation: An Office Environment for the Professional. *Behaviour & Information Technology*, 1(1):43–54.

Stachowiak, H. (1973). *Allgemeine Modelltheorie*. Springer, Wien.

Storey, M.-A. D. and Müller, H. A. (1995). Graph Layout Adjustment Strategies. In *GD '95: Proceedings of the Symposium on Graph Drawing*, pages 487–499.

Storey, M.-A. D., Wong, K., Fracchia, F. D., and Müller, H. A. (1997a). On Integrating Visualization Techniques for Effective Software Exploration. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 38 – 45.

Storey, M.-A. D., Wong, K., and Müller, H. A. (1997b). How Do Program Understanding Tools Affect How Programmers Understand Programs? In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 12–21.

Summers, K. L., Goldsmith, T. E., Kubica, S., and Caudell, T. P. (2003). An Experimental Evaluation of Continuous Semantic Zooming in Program Visualization. In *Proceedings of the IEEE Symposium on Information Visualization*, pages 155–162.

Tamassia, R., di Battista, G., and Batini, C. (1988). Automatic Graph Drawing and Readability of Diagrams. *IEEE Transactions on Systems, Man and Cybernetics*, 18(1):61–79.

Tan, D. S., Robertson, G. G., and Czerwinski, M. (2001). Exploring 3D Navigation: Combining Speed-Coupled Flying with Orbiting. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 418–425.

Tufte, E. R. (1990). *Envisioning Information*. Graphics Press, Cheshire, CT.

Tufte, E. R. (1997). *Visual Explanations: Images and Quantities, Evidence and Narrative*. Graphics Press, Cheshire, CT.

Ware, C. (2004). *Information Visualization: Perception for Design*. Morgan Kaufmann, San Francisco, second edition.

Ware, C. and Fleet, D. (1997). Context Sensitive Flying Interface. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, pages 127–130.

Ware, C., Hui, D., and Franck, G. (1993). Visualizing Object Oriented Software in Three Dimensions. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering*.

Ware, C., Purchase, H., Colpoys, L., and McGill, M. (2002). Cognitive measurements of graph aesthetics. *Information Visualization*, 1(2):103–110.

Xia, Y. (2005). *A Language Definition Method for Visual Specification Languages*. PhD thesis, University of Zurich.

Yoeli, P. (1972). The Logic of Automated Map Lettering. *The Cartographic Journal*, 9(2):99–108.

Yourdon, E. and Constantine, L. (1975). *Structured Design*. YOURDON Press, New York.

Zanella, A., Carpendale, M. S. T., and Rounding, M. (2002). On the Effects of Viewing Cues in Comprehending Distortions. In *Proceedings of the Second Nordic Conference on Human-Computer Interaction*, pages 119–128.

Curriculum Vitae

Name:	Tobias Emanuel Reinhard
Date of Birth:	April 27, 1980
Place of Citizenship:	Horw (LU), Switzerland

1987-1992	Grammar school, Solothurn (SO)
1992-2000	High school, Solothurn (SO)
2000	Military training school
2000-2004	Studies of computer science, University of Zurich
2005-2010	Assistant and doctoral studies at the University of Zurich